ABSTRACT

Title of dissertation: CROSS-LAYER CUSTOMIZATION PLATFORM FOR LOW-POWER AND REAL-TIME EMBEDDED APPLICATIONS

Xiangrong Zhou, Doctor of Philosophy, 2008

Dissertation directed by: Professor Peter Petrov
Department of Electrical and Computer Engineering

Modern embedded applications have become increasingly complex and diverse in their functionalities and requirements. Data processing, communication and multimedia signal processing, real-time control and various other functionalities can often need to be implemented on the same System-on-Chip(SOC) platform. The significant power constraints and real-time guarantee requirements of these applications have become significant obstacles for the traditional embedded system design methodologies. The general-purpose computing microarchitectures of these platforms are designed to achieve good performance on average, which is far from optimal for any particular application. The system must always assume worst-case scenarios, which results in significant power inefficiencies and resource under-utilization.

This dissertation introduces a cross-layer application-customizable embedded platform, which dynamically exploits application information and fine-tunes system components at system software and hardware layers. This is achieved with the close cooperation and seamless integration of the compiler, the operating system,

and the hardware architecture. The compiler is responsible for extracting application regularities through static and profile-based analysis. The relevant application knowledge is propagated and utilized at run-time across the system layers through the judiciously introduced reconfigurability at both OS and hardware layers. The introduced framework comprehensively covers the fundamental subsystems of memory management and multi-tasking execution control.

CROSS-LAYER CUSTOMIZATION PLATFORM FOR LOW-POWER AND REAL-TIME EMBEDDED APPLICATIONS

by

Xiangrong Zhou

Dissertation submitted to the Faculty of the Graduate School of the
University of Maryland, College Park in partial fulfillment
of the requirements for the degree of
Doctor of Philosophy
2008

Advisory Committee:
Professor Peter Petrov, Chair/Advisor
Professor Rajeev Barua
Professor Shuvra S. Bhattacharyya
Professor Bruce Jacob
Professor Chau-Wen Tseng

© Copyright by
Xiangrong Zhou
2008

Acknowledgments

During the years of my Ph.D. study, so many dramatic changes have happened in the school, in the outside world and in myself. I owe my gratitude to all the people who have made it possible for me to complete this journey.

First and foremost I'd like to thank my advisor, Professor Peter Petrov for giving me an invaluable opportunity to work with him on such interesting projects. I really appreciate all his time, financial support and the advises he has given me. It has been my great pleasure to work with and learn from Professor Petrov through the years of study and research.

I would also like to thank Professor Jacob and Professor Barua for giving me suggestions on my proposal. Professor Jacob's advises on academic career and Professor Barua's suggestions on presentations are all helpful to me. I also own great gratitude to Professor Bhattacharyya, Professor Qu, and Professor Jacob for agreeing to serve on my dissertation committee and writing reference letters for my academic job applications. I also owe my great thanks to Professor Tseng for spending his valuable time to serve on my committee as dean's representative.

I would also like to thank the ECE department for supporting me with the Teaching Assistantship in the first two years, and the graduate school for supporting me with the dissertation fellowship. Thank Professor Leandros Tassiulas, Professor John Baras and Dr. Michael Hadjitheodosiou for giving me financial support in the first two summers and leading me into the early research activities. I specially thank Dr. Dan Balon, Dr. Tracy Chung, Maria Hoo and Vivian Lu at graduate office for

helping me handle all the departmental rules and forms.

I would also like to acknowledge help and supports from my fellow graduate student colleagues. The collaboration with Mr. Chenjie Yu and Ms. Alokika Dash is really helpful to me on the multi-processor project I participated. I also appreciate the helps and advises from many former ECE graduates: Mr. Chang Wang, Dr. Hongjun Li, Dr. Zhu Han, Dr. Xiaojiang Du, Dr. Hong Zhao and Dr. Zhanfeng Yue, Dr. Hui Li, Dr Yupeng Cui and Mr. Qigong Zheng. Their help had made my early days of graduate study much easier. I also thank my soccer buddies at Maryland for bringing me so much fun of playing pickup games and tournaments together. I will definitely miss this part of my life the most after I leave Maryland.

I owe my deepest thanks to my family - my mother and father who have always stood by me and guided me through my career, and have pulled me through against impossible odds at times. Words cannot express the gratitude I owe them. Thanks go to my brothers who had encouraged me and given their best support through all the years.

It is impossible to list all, and I apologize to those I have inadvertently left out.

Table of Contents

List of Figures vi

1 Introduction 1
- 1.1 Motivation . . . 3
- 1.2 Hardware/Software Co-design . . . 6
- 1.3 Cross-Layer System Customizations . . . 9
- 1.4 Contribution of the Dissertation . . . 13

2 Related Work 17
- 2.1 Low Power Design Techniques . . . 17
- 2.2 Memory Management . . . 20
- 2.3 Multitasking Management . . . 24

3 Cross-Layer Customization for Memory Management 30
- 3.1 Direct Address Translation . . . 30
  - 3.1.1 Introduction . . . 30
  - 3.1.2 Tagless Direct Address Translation . . . 33
  - 3.1.3 Compiler and OS Support for DTT Management . . . 39
    - 3.1.3.1 VPN-Partition Formation Algorithm . . . 41
    - 3.1.3.2 Stack and Heap Memory . . . 49
    - 3.1.3.3 DTT Management and Multi-Tasking Environments 52
  - 3.1.4 Hardware Support . . . 55
  - 3.1.5 Analysis and Discussion . . . 59
    - 3.1.5.1 Real-Time Performance Improvements . . . 59
    - 3.1.5.2 Leakage Power . . . 61
  - 3.1.6 Experimental Results . . . 62
- 3.2 Heterogeneously Tagged Cache . . . 75
  - 3.2.1 Introduction . . . 75
  - 3.2.2 Heterogeneous Cache Tagging - A Functional Overview . . . 78
  - 3.2.3 Write-Back and Write-Through Caches . . . 84
  - 3.2.4 Identifying the References to Shared Memory . . . 87
  - 3.2.5 Low-Power Synonym Alignment . . . 91
  - 3.2.6 Hardware Support . . . 94
  - 3.2.7 Experimental Results . . . 98
- 3.3 Address Translation through Arithmetic Operations . . . 110
  - 3.3.1 Introduction . . . 110
  - 3.3.2 Arithmetic Address Translation . . . 112
  - 3.3.3 Compiler and OS Support . . . 116
  - 3.3.4 Hardware Support . . . 120
  - 3.3.5 System Analysis and Discussion . . . 122
    - 3.3.5.1 Multitasking Support . . . 122
    - 3.3.5.2 Dynamic Memory . . . 124
    - 3.3.5.3 Real-Time Performance . . . 124

3.3.6 Experimental Results . . . 125
3.4 Interval Page Table . . . 131
3.4.1 Introduction . . . 131
3.4.2 Motivation . . . 133
3.4.3 Page Tables Overview . . . 137
3.4.4 The Interval Page Table . . . 140
3.4.4.1 IPT Organization . . . 141
3.4.4.2 IPT Manipulation . . . 143
3.4.4.3 IPT Lookup . . . 146
3.4.5 Experimental Results . . . 151

4 Cross-Layer Customization for Multitasking Management 159
4.1 Introduction . . . 159
4.2 Motivation . . . 164
4.3 State Liveness and Preemption Deferral . . . 168
4.3.1 Register Liveness Analysis . . . 169
4.3.2 Switch-Points and Blocks . . . 172
4.3.3 Live State Preservation . . . 175
4.4 Compiler-driven Context Switch (CCS) . . . 177
4.4.1 Compiler and OS Support . . . 178
4.4.2 Hardware Support . . . 182
4.5 Register Mapped Context Switch (RMCS) . . . 185
4.5.1 Compiler and OS support . . . 186
4.5.2 Hardware Support . . . 189
4.6 Experimental results . . . 192

5 Compiler-driven Register Re-Assignment for Temperature Control 205
5.1 Introduction . . . 205
5.2 Related Work . . . 209
5.3 Motivation for Temperature-Aware Register Re-Allocation . . . 211
5.4 Temperature-Aware Register Re-Allocation Fundamentals . . . 215
5.4.1 Register Name Reassignment (RNR) . . . 218
5.4.2 Live Range Reassignment (LRR) . . . 220
5.5 Register Re-Allocation Algorithms . . . 224
5.5.1 Set-Partition Heuristic for RNR . . . 224
5.5.2 Live Range Reassignment Algorithm . . . 226
5.6 Experimental results . . . 229

6 Conclusion 241

Bibliography 243

List of Figures

1.1 Moore's law . . . 2

1.2 Feed-Back and Customization Design . . . 12

3.1 TLB-based hardware address translation . . . 33

3.2 DTT: Software-controlled address translation . . . 33

3.3 Set of consecutive VPNs . . . 35

3.4 Indexing SRAM table with LSBs . . . 35

3.5 Two VPN partitions, each indexed by 2 LSBs . . . 36

3.6 Mapping VPN partitions to DTT segments . . . 39

3.7 VPN partition merging . . . 43

3.8 Pseudo-code of the VPN-partition formation algorithm . . . 46

3.9 DTT index computation and DTT access . . . 57

3.10 Hardware architecture . . . 58

3.11 Energy comparison (normalized) for set-associative organizations . . . 73

3.12 Energy comparison (normalized) for fully associative organizations . . . 74

3.13 Using PID to eliminate aliasing with virtually tagged caches . . . 79

3.14 Superset bits - overlap between virtual index and VPN . . . 81

3.15 Conflicting cache indices for the case of synonym groups exhibiting identical physical tags and virtual superset bits . . . 81

3.16 Two processes sharing one data block; aligned virtual addresses . . . 83

3.17 Linear mapping from VPNs to PPNs . . . 90

3.18 Two processes sharing one data block; non-aligned shared virtual addresses . . . 92

3.19 The Synonym Offset Table (SOT) and the adders for converting superset bits and VPNs . . . 93

3.20 Overall hardware organization 97
3.21 A data object forming a VPN-segment 113
3.22 Multiple objects forming a VPN-segment 113
3.23 Multiple VPN-segments within a task 114
3.24 Compiler support and setup code insertion 117
3.25 Mapping load/store instructions to VPN-segments 119
3.26 SAT table with offsets for PPN computation 121
3.27 Address translation architecture 121
3.28 Energy reduction comparison 130
3.29 Virtual memory architecture 134
3.30 Traditional hierarchical page table organizations 137
3.31 Traditional inverted page table organizations 139
3.32 Page interval example 141
3.33 IPT: An entry per page interval 141
3.34 Adding an IPT mapping 144
3.35 Removing an IPT mapping 145
3.36 Hardware for IPT traversal 146

4.1 Context-switch mechanism for preemptive multitasking. 164
4.2 Hardware register renaming 166
4.3 Register live ranges and live state 170
4.4 Application “hotspot” with two *switch* points and blocks. 172
4.5 Register liveness for EJ 174
4.6 Register liveness for TRI 174
4.7 Switch-points/blocks placement 175

4.8 Structure of the CCS live state preservation. 180
4.9 CCS Hardware Support 183
4.10 RMCS methodology functional overview 186
4.11 Mapped register file organization 188
4.12 Mapped register file architecture 190

5.1 Thermal RC model 206
5.2 Histogram of register accesses; Scalar compiler optimizations 211
5.3 Histogram of register accesses; Aggressive compiler optimizations . . . 212
5.4 A loop Control Flow Graph (CFG) and Register live ranges 215
5.5 Register name reassignment 218
5.6 Live range reassignment (LRR) procedure 221
5.7 Overall algorithms for temperature-aware register reallocation 223
5.8 Pseudo-code for RNR set-partition heuristic 224
5.9 Pseudo-code for live range reassignment (LRR) procedure 226
5.10 Achieved register access distribution; Aggressive compiler optimizations 239
5.11 Achieved register access distribution; Scalar compiler optimizations . 239

Chapter 1

Introduction

Embedded systems have been extensively used in various markets. These systems can be found in household electronics, office equipments, handhold and mobile devices, and industry control systems. It is estimated that the embedded processors occupy more than 90% of entire processor market, far more than their general-purpose counterparts.

Moore's law, which states that the number of transistors integrated on a single chip doubles about every two years, has been holding true for the past forty years. As shown in Figure 1.1. Chips with billions of transistors are becoming into reality in the past two years. The previous board-level systems now are shrinking into chip-level systems, which are called System-On-a-Chip (SOC). The proliferation of nanometer technologies with ever decreasing geometry sizes has led to proportionally unprecedented levels of SOC integration. The large number of transistors and the high-speed clock frequency have provided powerful computation capability to the SOC platform.

As the direct result, more and more application tasks can be mapped into one SOC, which brings the system complexity to an unprecedented high level. The tasks include data computation, communication and signal processing, real-time control and various other types of applications, which show large diversity in terms of appli-

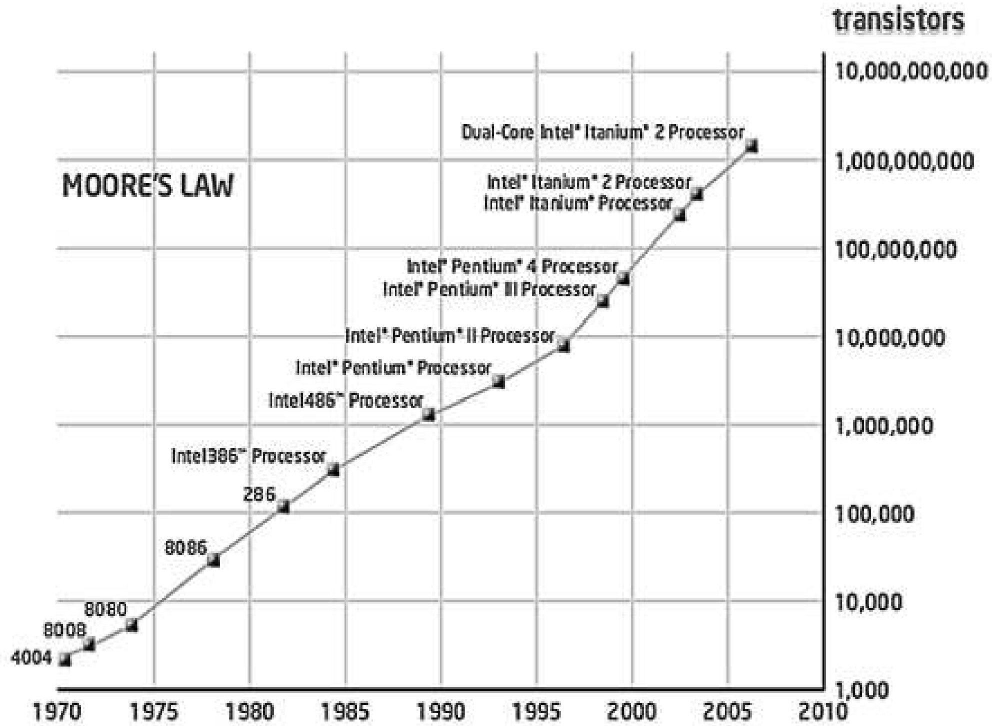


Figure 1.1: Moore's law

cation property and system requirements. The underlying hardware platforms are very difficult to be optimized for such diverse system requirements. In order to meet all the system performance specifications and efficiently utilize system resources, as well as fast time-to-market, lower design cost, and easier product maintenance, integration of multitude of embedded processor cores and other hardware accelerator blocks with hardware and software co-development have been adopted.

However, many embedded platforms still contain a large number of general-purpose mechanisms in the microarchitecture, system software and their interactions. Such generalities typically result in excessive power consumption, low performance, and pool real-time guarantee compared to a full-customized design. A

cross-layer system customization platform could dynamically integrate application information into system software and hardware platform, thus could fine tune the system to improve the performance and and resource utilization.

1.1 Motivation

Due to the large diversity of embedded system applications, the embedded system designers have to trade off the advantages and disadvantages between hardware and software design approaches. Many of the board-level system designs are combinations of an microprocessor with memory and several hardware logic circuits or peripherals.

The reconfigurable hardware logics dated back to PAL (Programmable Array Logic), where a number of OR gates and AND planes are used to implemented different combinational logic circuit [1, 2]. With the advances of the VLSI technology advances, CPLD(Complex Programmable Logic Device) and FPGA (Field Programmable Gate Array) are used to implement more complex functions in the hardware. Application Specific IC (ASIC) are fully customized with optimal performance, low cost and small hardware area. In the past forty years, the academic research and industry development on electronic design automation (EDA) have revolutionized the design process and significantly increased the hardware design productivity. The rich set of EDA tools basically cover almost every step of the hardware design flow including circuit analysis and simulation, synthesis, place and routing, testing and verification, packaging, etc. The development of hardware

description languages such as Verilog and VHDL makes todays digital hardware development somehow similar to software programming although the knowledge regarding the underlying hardware platform is still necessary.

However, even with such abundant tools, hardware implementation processes are still very expensive with the large initial development cost. The rigorous verification and testing step makes the development cycle rather long compare to software programming. The pure ASIC designs are not flexible for future design modifications and upgrades. On the other side, a hardware design has tremendous advantages of better performance due to parallelism, and lower power overhead and unit cost, compared to pure software design. Thus it is usually used in large volume and commodity or relatively mature applications.

In pure software design approaches, applications are implemented by means of software programs running on microprocessor cores. This processor-based design model is adopted from general-purpose processor system design. The overall system includes several layers: the hardware layer includes processor microarchitecture, memory and I/O devices; the operating system layer includes system functions such as process scheduling and memory management. It also transparently supports all the hardware resource sharing and provides a virtual machine interface to the applications; the application layer includes application programs and libraries, and the corresponding compiler's support. The academic research and industry development on design of microprocessor architecture and embedded operating systems have been very success. Today, the embedded processor commercial products range from very low cost processors like Atmel's 8051, AVR[3], to low-power ARM processor[4], to

relatively high-end PowerPC processor[5]. Special purpose processors such as Digital Signal Processor (DSP) and graphic processor (GPU) have very high performance for certain applications. For embedded operating systems, several products such as VxWorks[6], embedded Linux[7], ThreadX[8], WinCE[9] and etc. have different advantages and are being adopted by the application developers.

The advantages of processor-based software implementation is obvious because of this vertical layered system design and software implementation of user application: application software could be reused across platform, system integration and test are easier than hardware approach, future upgrade and maintenance are more flexible. However, even though most embedded operating systems are designed and implemented with consideration of real-time performance and memory constraints, the application performance and system power consumptions are still less optimized than ASIC implementation.

In todays highly competitive market, embedded applications need to be designed with multi-dimensional requirements in mind. On the business side, the design cost and final product cost are the most influential issues in the market with very low profit margins. Rapid time to market is also very important for the product to lead the trend on both the business and the technical sides. The flexibility of the selected implementation platform could enable new features to be added and thus bring extra profits. It also reduces the design reuse cost. On the engineering side, system performance and throughput are the most important design criteria since many modern embedded applications such as wireless communication and multimedia processing require large computing performance. Power and energy efficiency are

also becoming extremely important, if not more than performance, not only to mobile and hand-held device where battery life directly affects the product acceptance, but also to office and household electronics because of today's high cost of energy. Real-time performance as well as system reliability and security are also very important system requirements, especially for critical applications such as automobile control systems. The SOC (System-On-a-Chip) platform, which integrates multiple processor cores and hardware function blocks, has been tremendously advantageous due to the highly reduced cost, fast time-to-market, low energy consumption, stronger performance, higher reliability, and easier product maintenance. On the SOC platform, the microprocessor cores are either integrated with ASIC blocks or FPGA blocks. The processor cores can be standard processor core such as ARM or PowerPC, or customized such as Application Specific Instruction Processor (ASIP). They can come in the form of either a hard-core or a soft-core implemented with FPGA logic or synthesized beforehand for the specific process technology.

1.2 Hardware/Software Co-design

The embedded system design complexity combined with a very tight time-to-market requirements has revolutionized the embedded system design process. The concurrent design of hardware and software has displaced traditional sequential design. Furthermore, hardware and software design now begins before the system architecture (or even the specification) is finalized. System architects, customers, and marketing departments develop requirement definitions and system specifications

together. System architects define a system architecture consisting of cooperating system functions that form the basis of concurrent hardware and software design. Interface design requires the participation of both hardware and software developers.

The system integration and testing step consists of many individual steps. Reusing components taken from previous designs or acquired from outside the design group (Intellectual Property, or IP) is a main design goal to improve productivity and reduce design risk. A concurrent design starting with a partially incomplete specification requires close cooperation of all participants in the design process. Hardware and software designers and system architects must synchronize their work progress to optimize and debug a system in a joint effort.

Hardware/software co-design methodologies are critical in the design of such complex SOCs. The first challenge is modeling and partitioning system functionality between hardware and software. A major problem in the system partitioning process is synchronization and integration of hardware and software design. This requires permanent control of consistency and correctness. Hardware/software co-simulation is required to have different level of granularity since register-transfer-level (RTL) level hardware simulations are too slow for practical software simulation. The processor, memory and hardware modules must have different abstract execution models for the co-simulation.

The next challenge is hardware/software co-synthesis. Specialized architectures, such as DSP or micro-controllers, dominate the embedded processor market because of their cost and power efficiency. However, automatically generated software code is still not optimal compared to hardware solutions because of the large

overhead in the software implementation due to the microarchitecture complexity and the memory wall. Porting functions between hardware and software implementation is also very difficult. Many high level synthesize tools such as MATLAB or SystemC are developed and suitable for simple architectures and low performance requirements.

Hardware/software co-verification is even more cumbersome as the test space grows exponentially. The software and hardware modules impose different testing granularity and requirements. Automatic and reliable system level testing, especially testing the communication interface between hardware/software boundaries and processor/processor boundaries are still not mature enough for complex application cases.

In overall, hardware/software co-design has made considerable progress in the past few years. Co-simulation, co-synthesis, and co-verification for the SOC development have large advantages in terms of cost, performance, power consumption and reliability, compared the the board-level system design with multiple discrete chips.

However, due to the heterogeneous application task and heterogeneous hardware component that are mapped to the SOC, the system level management of the software tasks and hardware component in the complex and dynamic run-time environment are even becoming challenging. The existing system resource management techniques are mostly inherited from general-purpose systems. Many of the system resource management schemes such as memory management, multitasking management are not optimized for these complex SOC platforms, and there exist

large system level performance overhead and power over-consumption.

In the next generation of SOC platform, there is strong need to new system resource management techniques that could efficiently handle such complex and dynamic embedded systems. One way to achieve such intelligent management schemes is to dynamically fine-tune the system components to the application property and system requirement in the runtime, so that the overall system performance is improved and power consumption reduced with little system overhead, which motivate my research topic of the cross-layer system customization for low-power and real-time embedded applications.

1.3 Cross-Layer System Customizations

Fundamentally, a general-purpose processor architecture is a universal computation engine with a micro-architectural design targeted to achieve good performance and power characteristics *on average* across a large range of possible programs. While good on average, these micro-architectural components are far from the optimal performance and power points for any particular application. Furthermore, system softwares must always assume worst-case scenarios in terms of system-level information. For instance, to meet the real-time requirement, all tasks are scheduled assuming worst time execution time(WCET). For specific application, this over design under worst-case assumption would result in many waste utilization of system resources.

In embedded processors which adopt super-scalar micro-architecture, program

properties such as branches, data reuse are typically dynamically predicted by run-time traces. Due to limit hardware resource availability and program intrinsic complex logic, this prediction could result in poor estimation of the the real program parameters while adding substantial power consumption.

Certain domain specific processors in which the micro-architecture are designed to focus on certain domains of applications such as digital signal processing, networking switching. Such processors support many general-purpose type of instruction set with enhance of dedicated instruction and micro-architecture support optimized for the representative applications from this domain. For example, as audio/video compression standard typically utilized various DSP algorithm such as FIR/IIF filtering or FFT transformation, a DSP micro-architecture typically includes bit-reverse register addressing and circular buffer addressing mode to speedup the memory access. In high end DSP and graphic processor which utilize VLIW(Very Long Instruction Word) architecture, many Single Instruction Multiple Data (SIMD) instructions are included in the ISA and very complex complier techniques are designed to explicitly perform Instruction Set Parallel (ILP) to speed up the performance.

To further improve the performance for specific application, a type of Application Specific Processor (ASP) has been developed in the past few years. In typical implementation of an Application Specific Instruction Processors (ASIP), a baseline processor micro-architecture is used that can be automatically enhanced with customized instructions to significantly improve performance. Some FPGA-based ASPs have been proposed in [10], where the whole processor core can be reprogrammed

at any time for the target application requirements, including the instantiations of customized pipelines.

The lack of deterministic application knowledge within the pre-designed systems is the fundamental difficulty in such types of system optimizations. If such application specific information can been extracted and provided to the run-time system, the overall system performance and efficiency can be significantly improved. The compiler could utilize the application information to generate optimal code, the operating system can optimally schedule the tasks for execution and perform context switches efficiently. The micro-architecture can be designed and fine-tuned so that power efficiently is significantly improved. As shown in Figure 1.2, this customization includes the feed-back steps and the customization through all the systems layers including application, operating systems and hardware architecture. To fine-tune the system in the run-time, the feed-back path will extract specific application information either through profiling or run-time monitoring of system properties. The application information is analyzed and partitioned into customization module and distributed into each layer depending the targeting optimization objectives. This could be inserting certain probe code in the binary, or parameter tables to reconfigure the hardware blocks.

With the cooperation of complier, operation system, and hardware architecture, the proposed cross-layer application customizable platform dynamically exploits application information and fine tunes the system software and hardware platform. The compiler is responsible for extracting application regularities through static and dynamic analysis, especially within hot-spot regions which are the most

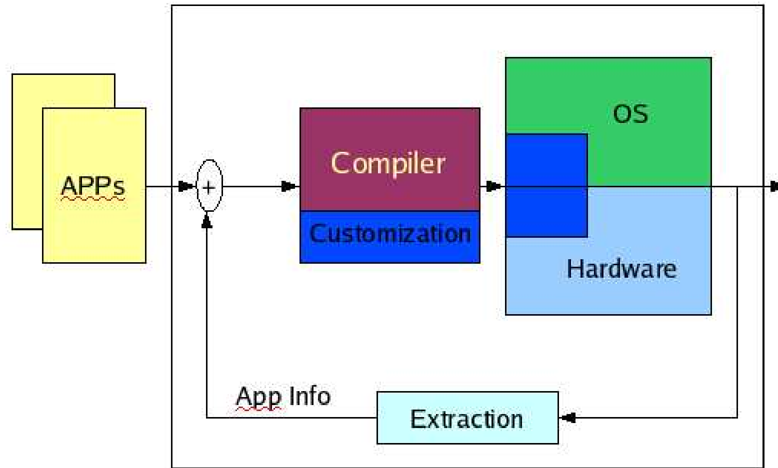


Figure 1.2: Feed-Back and Customization Design

frequently accessed parts of the application program and where computation is most intensive. A well known rule-of-thumb in computer architecture is that a program typically spend 90% of its execution cycles only in 10% of its code. Such hot-spots usually consist of frequent executed loops or functions, such as various algorithmic (numerical and signal processing) kernels. Because such hot-spot regions of code also potentially contain more regularity, an efficient customization of these hot-spots can significantly improve the performance and energy efficiency.

The system functions within the OS not only can be dynamically adjusted to application requirements, but also in their interaction with both the application and the hardware layers. At hardware architecture level, the right kind of reconfigurability has to be incorporated, since allowing complete FPGA-like reconfigurability would lead to a significant power and performance overheads. The right level of programmability must be identified so that all the relevant application properties

and regularities can be efficiently captured with minimal power, performance, and area impact. This can be efficiently achieved since most of the relevant application information regarding architectural components like caches and TLBs can be easily represented in a very regular way.

1.4 Contribution of the Dissertation

The objective of my Ph.D. research is to investigate application customization techniques and incorporate them into a system-level customization platform. The targeted sub-systems that we focused are mainly complex system management services, such as memory management and multitasking management, which typically require system-level information and could not be optimized by software or hardware alone. The substantial customization method will discussed and implemented in simulation tools. Then the overall experiment will by collected and evaluated in terms of performance improvement and power reduction.

Virtual memory has been adopted in general-purpose system for abstraction of memory management, which transparently provides services such as memory reallocation, data sharing and memory protection to application programs. The general-purpose design of virtual memory support requires the cooperation of system software which maintains page tables that hold all the translations mappings between each virtual address and its corresponding physical address, and a hardware cache(TLB, Translation Lookaside Buffer) which caches the most frequently used translation entry. However virtual memory would be beneficial to embedded

system, the non-deterministic TLB access time due to the frequent TLB misses or even page table lookup misses, as well as the excessive energy consumption overhead of TLB hardware accessed in every memory access, have been major obstacles for many embedded processors to support virtual memory in multitasking systems.

I have investigated several customization techniques to facilitate the adoption of virtual memory in low-power embedded systems. Similarly to scratch-pad memory(SPM,on-chip SRAM) to replace cache for low-power processor, Direct Translation Table (DTT, a small on-chip SRAM) was proposed to replace TLB to provide deterministic and low-power translation access [11, 12]. Application address mapping information are collect and merged into compact partitions to share the DTT with high utilization. The application and OS work together on prefeteching and sharing partitions on the DTT. When the partitions of consecutive memory block such as large data array or task stacks are mapped consecutively in physical space, the physical page translation procedure is logically adding an offset to its corresponding virtual page number. For such special case, the table lookup can then be reduced to an simple parallel adder hardware to reduce power consumption [13, 14]. In general-purpose processor with cache, physical address translation is necessary for every memory access as virtual addressing could possible result in cache aliasing problem when same virtual address in different applications are mapped to different physical location, and cache synonym problem when different virtual addresses are mapped to the same physical address (caused by data sharing). Extending virtual address with process ID could resolve the former, we investigated the technique that embeds application sharing information in instructions and only translate physical

address only for possible shared data with the help of hardware differentiating cache accesses [15, 16]. A special structured page table that compresses multiple translation entries of the same consecutive mapped segment into one wider entry was investigated for embedded systems where memory remapping are very rare cases [17].

To respond to frequently-triggered asynchronous events, preemption based multitasking systems are adopted in many real-time embedded systems. Each switch involves large number of memory accesses, as operating system will first save the context of current task(preempted task) and load the context of next highest-priority ready task(preempted task). The high frequency of context switches not only affect the throughput of system effective instruction, but also increase each task's response time. By analyzing the live register utilization through CFG(Control Flow Graph)'s paths and enabling switches at the point with minimal live sets, we effectively reduce necessary live context size. Deferring switches to nearest minimal point by customized hardware and preserving the minimal live set by customized OS callback functions enhance promptness of preemption, and increase the effective instruction throughput as useful application instruction are still being executed during deferral [18, 19]. To further reduce memory accesses of preserving contexts, a small pool of spare register pages are introduced to immediately remap live context [20]. The page level remapping is done through simple hardware structure, while the live registers are packed into small regular pages after the compiler's register allocation phase.

Temperature hot-spots have been known to cause severe reliability problems

and to significantly increase leakage power. Due to frequent access and relatively small area, the register file has been previously shown to exhibit the highest temperature in many modern high-end embedded processor, which makes it particularly susceptible to faults and elevated leakage power. We show that this is mostly due to the highly clustered register file accesses where a set of few registers physically placed close to each other are accessed with very high frequency. A compiler-based register assignment methodology, which purpose is to break such groups of registers and to uniformly distribute the accesses to the register file is investigated [22].

Chapter 2

Related Work

2.1 Low Power Design Techniques

The overall chip power consumption as the sum of dynamic and static power can be expressed by the Equation 2.1 [23]:

$$\mathbf{P} = A \cdot C \cdot V^2 \cdot f + V \cdot I_{leak} \quad (2.1)$$

The first term is the dynamic power lost from charging and discharging the processors capacitive loads: A is the fraction of gates actively switching and C is the total capacitance load of all gates. The second term models the static power lost due to leakage current, I_{leak}. The power lost to the momentary short circuit at a gates output is ignored here since the loss is relatively small; it contributes to dynamic power loss, and the equations first term can absorb it, if necessary.

When dynamic power is the dominate the leakage power as it has been and as it remains today in many less aggressive fabrication technologies, Equation 2.1 can be approximated by the first term. Reducing the supply voltage with the factor of V_2 is the most effective way to decrease power consumption. Scaling down the clock frequency, using low power transistors, and reducing switching activity will reduce f, C and V respectively so that the dynamic power can be reduced proportionally.

Leakage current, the source of static power consumption, is a combination of

sub-threshold and gate-oxide leakage: $I_{leak} = I_{sub} + I_{ox}$, where sub-threshold current and gate-oxide current can be expressed by Equation 2.2 and Equation 2.3:

$$I_{sub} = K_1 \cdot W \cdot e^{-V_{th}/nV_\theta} \cdot (1 - e^{-V/V_\theta}) \tag{2.2}$$

where K_1 and n are experimentally derived, W is the gate width, and V_θ in the exponents is the thermal voltage. At room temperature, V_θ is about 25 mV; it increases linearly as temperature increases. If I_{sub} grows enough to build up heat, V_θ will also start to rise, further increasing I_{sub} and possibly causing thermal runaway. Turning off the supply voltage could sets V to zero so that the factor in parentheses also becomes zero. Increasing the threshold voltage can have a dramatic effect of reducing the sub-threshold current in even small increments.

$$I_{ox} = K_2 \cdot W \cdot (V/T_{ox})^2 \cdot e^{-\alpha T_{ox}/N} \tag{2.3}$$

where K_2 and α are experimentally derived. T_{ox} is oxide thickness. The research and development of community high-k dielectric gate insulators is to reduce this leakage current.

The direct method to reduce the power consumption is to target parameters at circuit level. In DVFS(Dynamic Voltage and Frequency Scaling) design, the supply voltage and the clock frequency are reduced to at minimum level when there is no critical task running. Various DVFS techniques have been both proposed in academia[24, 25] and developed in industry [26, 27]. Find appropriate transistor size [28] or redesign complex gate[29] will change the load capacity or reduce the

total number of transistor count, thus reduce the power consumption. Clock gating technique which pauses the clock input to the synchronous transistors when the function units and register files in the system are not active in the near future could significant reduce the power consumption [30].

Architecture level power reduction focus on larger granularity of power management. The power overheads of each part are analytical profiled to accurately estimated the power distribution in the system. Various power analysis and estimation techniques has been discuss in [31, 32]. The next step is to either replace that unit with alternative low power architecture design [33], [34]. Low power cache architectures have been proposed in [35], [36] as well.

Interconnect power reduction techniques focus on the reducing redundant power consumption in inter module communication activities in the SOC. Code compression techniques which compress the code before transit to bus and decompress after received from bus could significant reduce interconnect activity although there is some compression power overhead[37]. Bus encoding techniques in which the context of the data and frequency of each data is analyzed, the data transmitted on the bus is encoded to a power efficiently format and decoded at the other end of bus. Thus per activity cost is saved[38].

The software level or system-level power management techniques focusing the efficiency of utilizing the hardware components. For example, the compiler could analyzed the memory ambiguity and save the memory load/store activity[39]. The power-aware scheduling approach in [40] combined the DVS technique and OS scheduling algorithm to reduce the overall run-time power consumption. In [41],

the author applied the application memory access information to the compiler and the cache architecture to reduce the total cache access energy.

2.2 Memory Management

The memory subsystem has been known to be one of the major bottlenecks in terms of power and performance not only for general-purpose computing systems but also, and even more so, for the typically resource constrained embedded systems [42, 43].

Virtual memory [44, 45] has been well known as an elegant approach to abstract from the application the complexity of *memory allocation*, and *code/data relocation and sharing*, while efficiently providing *memory protection* between user applications and system software; all these being completely transparent to the application and controlled by the operating system (OS). Such features would tremendously benefit many embedded systems, if virtual memory is to be supported for them. In recent years many high-end embedded processors have started to employ a *Memory Management Unit (MMU)*, such as Intel XScale [46], ARM720 [47], and ARM9 [48]. The MMU is a hardware structure responsible for translating addresses generated by the processor to physical memory addresses. General-purpose virtual memory, however, requires unacceptably high amounts of power and introduces execution time nondeterminism, thus rendering itself unusable for a large number of embedded applications with stringent power constraints and real-time requirements.

When virtual memory support is present, the program accesses a virtual ad-

dress space partitioned into pages, which are referred to as *virtual pages* and are identified by their *Virtual Page Number (VPN)* which constitutes a large fraction of the virtual address most significant bits. During each memory access a translation is needed to map the virtual address into a physical one. The translation is performed at a page granularity in order to control the complexity of the translating mechanism. The *Translation Lookaside Buffer (TLB)* is a hardware cache responsible for capturing the most recently used *Page Table Entries (PTE)* for dynamic virtual address translation with no intervention of the system software. The mapping between virtual and physical addresses is typically maintained by the OS and established by the OS loader, dynamic linker and memory manager. TLB misses typically result in trapping into the OS where the missing PTE is retrieved from the *page table* maintained by the kernel. As this process is complex and time consuming, the TLB is usually implemented as a highly associative cache structure so that misses are minimized, which, in turn, results in significant amount of power consumption. In [49] it has been demonstrated that the TLB power constitutes 20%-25% of the total cache power consumption, which in turn has been shown to comprise in some cases around 50% of the total chip power [50]. It has been shown through direct measurements [51, 52] that around 17% of the total on-chip power for the StrongARM and the Hitachi SH-3 is contributed by the TLB.

The need for energy costly address translation on each memory reference, as well as the introduced execution time uncertainty caused by the cache-like TLB lookups, have been the two major factors preventing the introduction of virtual memory and its benefits to low-power and real-time processors.

The importance of the TLB in terms of performance and power has been recognized in the microarchitecture industry and research communities. Consequently, techniques for minimizing the power and performance overhead of TLBs have constituted the focus of a number of research activities in the recent years. A low-power TLB organization for chip-multiprocessors has been proposed in [49]. By incorporating a special *Page Sharing Table* to the TLB and using virtual caches, the authors reduce the amount of TLB activities, at the same time eliminating a large number of snoop accesses. A similar work in the direction of employing virtual caches with specialized TLB support is presented in [53]. The authors propose replacing the TLB with the more scalable and power efficient *Synonym Lookaside Buffer*, as it stores only the current synonym instances. In [51], the authors evaluate the power consumption of a number of TLB organizations and propose a new cell implementation for low-power set-associative TLBs. A low-power and high-performance TLB architecture has been proposed in [54]. The result of each TLB lookup is latched and prior to accessing the TLB the previously latched address translation is first looked-up, thus eliminating the TLB access if that same virtual page has been accessed in the previous memory reference. The concept of synonymous translations has been introduced in [55]. Superscalar processors can execute multiple memory references per cycle, out of which many refer to identical virtual pages; such synonymous TLB accesses within a cycle and across cycles are identified and the TLB lookups are compacted to the minimum needed. In [56] the TLB and cache accesses are partitioned according to their semantic such as static, global, stack, and heap data. The unique behavior and locality of each reference partition is exploited and each

stream redirected to its micro-TLB. A recent approach [57] proposes reconfigurable decoder structures for generating cache indices for direct-mapped cache structures. Cache line utilization is balanced and large number of conflicts eliminated thus approximating the functionality of highly associative caches yet at much lower power consumption. Another recently proposed approach for minimizing cache indexing conflicts was described in [58]. The cache index in this approach is computed by selecting a certain number of bit positions from the address. Which bits positions to select is determined during compile-time and a special hardware structure is configured prior to executing the program so that the cache index is formed by selecting the identified bit cluster. A TLB organization is proposed [59] that dynamically supports up to two pages per entry with a banked fully-associative structure. Such an organization benefits applications where larger pages can be used to minimize the translation overhead. In [60], the TLB accesses are redirected to a register file which holds a few recent TLB entries. Due to the small size of the register file compared to the VPN footprint, the compiler needs to reconstruct the code in order to minimize the overhead of replacing register entries at run time. Aspects of virtual memory have been modeled in software through a compiler inserted code in the application. Such an approach, where memory protection is implemented by the compiler is proposed in [61].

When virtual memory and caches coexist, techniques for power reduction and performance improvements for caches and TLBs together needed to considering the overall optimization. Cache conscious memory layouts have been explored for both code [62, 63] and data [64, 65]. In [66], the authors have proposed the *U-cache* archi-

tecture which maintains a reverse translation information of the cache blocks that belong to unaligned virtual pages only in order to handle the synonyms efficiently. A low-power physically-tagged cache has been proposed in [67]. A minimal set of tag bits is dynamically identified per hot-spot and used to access the cache instead of complete tags. In [52] the authors have proposed an instruction address translation architecture, which places the most recent I-TLB translation in a register, which is subsequently being reused until the instruction memory page is changed. For periods of time when instructions are fetched from the same memory page, the translation register is used to obtain the physical address instead of the I-TLB thus achieving faster and less power consuming instruction address translation. Compiler techniques have been presented in [68], which maximize the reuse of the software-controlled translation registers. Methodologies which emulate the address translation process in software have been recently proposed in [69]. In these approaches the compiler introduces code for run-time checks in order to enable applications to share physical memory for their stacks and to prevent any out-of-boundary memory accesses.

2.3 Multitasking Management

Many modern embedded applications, such as personal organizers, cell phones, and various hand-held devices, constitute complex computing systems where multiple execution tasks cooperate in implementing the product specification. Due to market demands, a large number of capabilities need to be supported, such as

aggregated multimedia data processing (speech, audio, video), communication protocols (GSM/CDMA, VoIP, Bluetooth, CAN), security functions, user interfaces, and many others. The utilization of embedded processors for real-time and time-critical control applications have been growing rapidly. The modern automotive industry, for instance, has adopted the approach where tens to hundreds of such processors are used throughout a single automobile [70]. They are used for traction control, anti-lock brake systems, engine control, and many other control and time-critical tasks. Many real-time data acquisition and processing systems such as sensor nodes and networks, impose strict real-time constraints and response time in order to capture, process, and identify rapidly appearing objects and physical phenomena. At the same time, all this processing power needs to be achieved with extremely energy-efficient and low-cost embedded processors.

The inherent multi-tasking nature of these applications has led to implementations where multiple software tasks are mapped for execution on a high-performance embedded processor such as the Intel XScale [46] and the ARM9 [48], which offer multi-tasking support in the form of *MMUs* and *hardware timers*, and readily available operating systems (OS) which utilize this hardware to implement various forms of multi-tasking.

The two widely adopted schemes for task switch control are the *cooperative* and the *preemptive* multi-tasking. In cooperative multi-tasking, the task voluntarily releases the control of the CPU to the OS at certain points of its execution. This release typically occurs when the task finishes execution or when the task computation load is low and is waiting for a lengthy I/O operation. Such an approach is

followed in TinyOS [71] where tasks are executed in a manner of run-to-completion. In this approach, longer tasks need to be partitioned into shorter ones. As pointed out in [72], such run-to-completion scheme can cause problems with meeting real-time constraints, as it is not possible to partition many tasks, which can result into a situation where a single task occupies the CPU for a long time. The cooperative multitasking paradigm is further explored in [73], where the authors have proposed to integrate multiple threads into a single thread statically during compile time. The benefits of cooperative multitasking for networking applications have been analyzed in [74]. Even though these approaches have the advantage of avoiding nondeterministic context switch overheads, an extra limitation on the dynamic behavior is created as they require that all preemption points are known during compile time. The degraded responsiveness to asynchronous events has been the major disadvantage of cooperating multitasking.

In preemptive multi-tasking the OS can pause a low-priority task and assign the CPU to a higher priority task - an OS controlled event referred to as *preemption*. Preemptive multi-tasking relies on a timer to generate interrupts at regular time intervals. When such an interrupt occurs, the execution control is transferred to a kernel routine that determines whether a task switch needs to be performed and, subsequently, to perform the context-switch. As this approach has the distinctive advantage of better responsiveness and stability, most of real-time scheduling algorithms and OS multitasking support are based on it [75, 76]. However, the frequent preemptions interrupt the normal task execution and bring extra performance and power overheads in the form of cycles needed to preserve and then restore the task

context. The task context includes the entire register file and all the status registers such as the Program Counter (PC) and the Stack Pointer (SP); its size is by far dominated by the register file. As task preemption exhibits asynchronous behavior, the OS kernel must be conservative and preserve/restore the entire state.

Due to cost and power constraints the majority of modern embedded processors follow the RISC and VLIW paradigms. In these architectures, and even more so in VLIW, the register file is traditionally very large in order to enable aggressive compiler optimizations targeting instruction parallelism and execution throughput. A typical modern VLIW architecture [77] features a general-purpose register file of size from 64 to 256. It has been shown [78] that for some short tasks responsible to react and process data samples in sensor networks, the context switch overhead can be up to 30% of the total execution cycle.

Instead of having a distinct physical register file for each task, another hardware solution is to have a relatively small ISA-visible set of registers, while implementing a significantly larger physical register file. This organization is illustrated in Figure 4.2. At run-time, each virtual register is renamed to a free physical register. This approach is very popular in superscalar processors, such as Intel Pentium 4 [79] and Alpha 21264 [80]. Such hardware register renaming is mostly used to exploit the available ILP in the program. Context switches are fast as typically only a small part of the physical register file needs to be preserved in memory. However, due to its per-register granularity and the fact that the renaming hardware needs to be activated at every cycle, the approach suffers from excessive power consumption and as such is not applicable to embedded systems. With a similar objective, in [81]

the physical register file is implemented as a cache that captures a large number of virtual registers. Fast access to subroutine and multithread contexts is achieved with a non-trivial power overhead.

The notion of fast context switch point has been first introduced in [82]. Each instruction is marked with a special bit to indicate whether a fast context switch is possible at that point. A fast context switch point is defined as an instruction where all scratch registers are dead. Scratch registers are a subset of all the registers which are caller-saved across function call boundaries; the context switch mechanism saves and restores all the remaining non-scratch registers. Consequently, this is a "all-or-nothing" approach targeting old architectures with rather small register files and no register windowing. VxWorks [6], on the other hand, provides a special hardware context for interrupt service code in order to avoid preserving the task context, and thus improving responsiveness to various system generated events. In [83], the authors have proposed a Simultaneous Multi-Threading platform with mini-thread execution. This approach, however, introduces a non-trivial hardware overhead. In [84], the authors have proposed to reduce the task context in the static OS by finding the live set of each task and merge the set by using the preemption priority information. The authors in [74] utilize and explore cooperative multi-threading instead of asynchronous preemption. Other research has shown that for some applications with known set of tasks and well known run-time characteristics and interactions, an efficient cooperative multitasking system can be synthesized through software thread integration [85, 86]. In a more dynamic system, however, with preemptive multitasking, the active task may have to be suspended at arbitrary point so that

another task is placed for execution. Even though the task switch overhead is reduced, the system responsiveness is limited as the compiler must statically decide on the way tasks are interleaved.

Chapter 3

Cross-Layer Customization for Memory Management

Virtual memory [44, 45] has been well known as an elegant approach to abstract from the application the complexity of *memory allocation*, and *code/data relocation and sharing*, while efficiently providing *memory protection* between user applications and system software; all these being completely transparent to the application and controlled by the operating system (OS). Such features would tremendously benefit many embedded systems, if virtual memory is to be supported for them. In recent years many high-end embedded processors have started to employ a *Memory Management Unit (MMU)*, such as Intel XScale [46], ARM720 [47], and ARM9 [48]. The MMU is a hardware structure responsible for translating addresses generated by the processor to physical memory addresses. General-purpose virtual memory, however, requires unacceptably high amounts of power and introduces execution time nondeterminism, thus rendering itself unusable for a large number of embedded applications with stringent power constraints and real-time requirements.

3.1 Direct Address Translation

3.1.1 Introduction

In general-purpose architecture designs, the assumption is that a large variety of programs are to be executed and that there is no program information

made available to the microarchitecture prior of its execution. It is also assumed that the program to be executed could come in a binary only form. Embedded processors and systems, however, have the distinctive advantage of complete application knowledge, as the embedded software is usually developed concurrently with the hardware design or is available in a source code format. The energy-efficient and time-deterministic address translation schemes outlined in this section are techniques that with the help of the compiler and the operating systems exploit dynamically such application-specific knowledge.

Through the utilization of application-specific information regarding the virtual memory footprint of the application, the set of VPNs accessed by the program is *partitioned into groups* so that by using only a small number of least significant VPN bits as an index into a special hardware translation table, *a conflict-free, and thus tag-less lookup* can be achieved. This not only results in very low-power address translation, but also to highly predictable execution times as the conflict free *Direct Translation Table (DTT)* access guarantees *fast* and *time-deterministic* address translation with no intervention of the OS for the majority of load/store instructions. This property is of great importance for real-time applications, where a worst-case execution time analysis of the program code is performed in order to guarantee the completion of certain computation within the pre-specified time deadlines.

The proposed methodology relies on the combined efforts of compiler, operating system, and hardware architecture to achieve both significant power reduction and deterministic address-translation times. It has been shown that any application program spends most of its execution times in a few relatively small parts of its code,

typically corresponding to loops or functions. Such parts of the code are usually referred to as *phases* of *hot-spots* [87, 88]. By targeting the application hotspots, practically all the benefits from the proposed technique can be achieved with only a low-cost hardware support needed to capture the information regarding the memory utilization. Consequently, the proposed techniques are applied on the application hot-spots, while for the rest of the infrequently executed part of the applications, the generic address translation mechanism is used. Upon entering or exiting a hot-spot, the compiler inserts a special setup code which stores certain information into special registers and tables implemented as a part of the specialized hardware support and, thus, informs the hardware that a hot-spot has just been entered.

The utilization of TLB for address translation is analogous to the cache utilization in embedded processor, where the tag operations introduce significant power consumption and non-deterministic access time due to possibility of conflicts. In many real-time systems, embedded processors with scratch-pad memory are rather used. Scratch-pad memories are SRAM memories mapped into the physical address space of the processor. Compared to caches scratch-pad memories provide for deterministic performance and energy efficiency. Various software controlled schemes for scratch-pad memory have been proposed and proved effective for data caching functionality [89, 90, 91, 92, 93], providing for fast and low-power data access. The large or difficult to allocate in scratch-pad data arrays can still utilize the services of hardware data cache. The mechanism replaces the majority of TLB cache-like accesses with software-managed direct-indexed SRAM table, thus offering energy-efficient TLB functionality with deterministic translation times. By effectively utilizing ap-

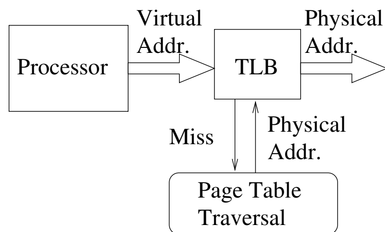


Figure 3.1: TLB-based hardware address translation

	Hardware	Software
Data Access	Data Cache	Scratchpad Memory
Address Transl.	TLB	*Proposed DTT*

Figure 3.2: DTT: Software-controlled address translation

plication information regarding the virtual pages used in the address translation process, a direct table with no tag arrays is used to avoid the energy overheads and nondeterministic times caused by conflicts in the TLB. In contrast with the other approaches, the technique that we propose does not trade-off performance for power. On the contrary, the proposes technique as it eliminated accessed to the TLB for most of the VPN accesses, it drastically reduces the number of TLB misses and thus slightly improves performance and significantly improves the execution time predictability - a characteristic of great importance to any real-time embedded system.

3.1.2 Tagless Direct Address Translation

The role of the TLB is to cache the most frequently requested virtual to physical address translation. After a virtual address is generated by the processor, the

TLB is looked up in order to determine the physical address as shown in Figure 3.1. In the case of a TLB miss, a page table traversal procedure is activated in order to find the correct translation. The page table traversal can be implemented in either hardware or software. The TLB is a cache-like structure, which captures the most frequently requested page table entries. Because of its cache-like organization and possibility of conflicts, the TLB contains tag arrays which are read and compared against the accessed VPN to determine whether the TLB look-up is a hit or a miss. The possibility of conflicts and the existence of the tag arrays and tag operations are the reasons for the excessive power consumption and the difficulty in estimating prior to program execution whether a TLB access will hit or miss [94, 95]. Clearly, this situation is very similar to data caches and their high power consumption and difficulty in statically analyzing their behavior. To resolve these issues of data caches, scratchpad memories have been used when deterministic execution times are needed together with low-power requirements. The proposed DTT-based address translation scheme is somewhat analogous to scratchpad memories for data caching as illustrated in Figure 3.2. The DTT is software-controlled and the translation entries are allocated into it in a way, which enables direct indexing, hence provide for energy-efficient and time-deterministic address translation.

The set of virtual pages allocated to the application code and static data is available after compiling and linking the program. The starting virtual address of the stack data is available as well; if memory is dynamically allocated, the particular VPNs are determined and known to the OS memory manager. The proposed address translation scheme exploits this information available to the compiler/linker and to

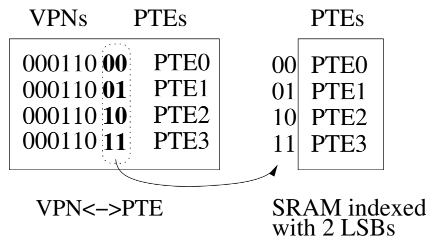


Figure 3.3: Set of consecutive VPNs

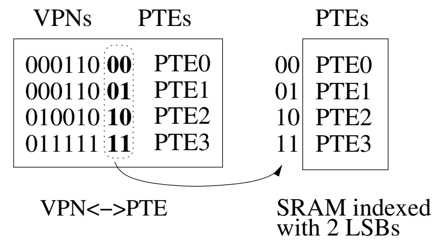


Figure 3.4: Indexing SRAM table with LSBs

the operating system in order to judiciously allocate the translation entries into the DTT. The DTT is subsequently accessed in a manner similar to a scratchpad memory.

If the possibility of TLB conflicts can be avoided through a judicious analysis of the VPN set, then a direct indexing for finding the PPN, free of any tag operations, can be achieved. Figure 3.21 shows an example where the compiler/linker has identified that only four consecutive data virtual pages can be accessed throughout the subsequent program execution. It can easily be seen that among all the VPN bits, the two *Least Significant Bits (LSBs)* are enough to differentiate the four VPNs. Consequently, these two LSBs can be used as an index into a translation table, where the VPNs will be mapped into the table in a conflict-free manner. It is evident that for any set of n consecutive VPNs, the $\lceil \log_2 n \rceil$ LSBs can be used as an unique identifier for any of the VPNs and as such they can be used as index to a table that stores information for any of the VPNs. A slightly more complex example is illustrated in Figure 3.22, in which four distinctive and not consecutive virtual pages are accessed. Although they are not consecutive, their two LSBs are still enough to uniquely distinguish them. Consequently, only these two bits of the

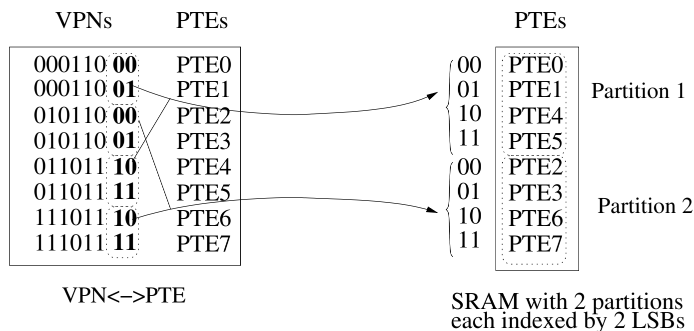


Figure 3.5: Two VPN partitions, each indexed by 2 LSBs

VPN can be used to form an index into a 4-entry memory block which holds the physical page addresses of these virtual pages, as shown in the figure. By avoiding the VPN tag look-up and using only these two bits as an index there would be no performance implications, while the overall reduction on the TLB power consumption is to be quite significant. All the power associated to the VPN tag arrays, the corresponding sense amplifiers, and the comparator cells is eliminated.

In the above examples not only is the power reduced but also the address translation timing for these VPNs becomes completely deterministic as by allocating the translation information for each VPN at the appropriate location in the table it can be guaranteed that the translation for these VPN will always complete within a cycle. This timing determinism is extremely important for real-time application where worst-case performance analysis need to be performed in order to guarantee that a certain processing is completed within a pre-specified time deadline.

The fundamental idea of the proposed approach is to identify such a conflict-free indexing scheme and to ensure a conflict-free allocation of the translation information in order to avoid the power consuming VPN tag operations and to provide

for fast and deterministic address translation times. Given a set of n data VPNs, there exists the minimal number m of VPN LSBs that could differentiate these VPNs and thus be used as an index. Even more importantly, how efficiently will the introduced translation table be utilized after storing the translation entries of each VPN in such a 2^m sized memory. The above examples show ideal situations where a minimal number of index bits are used and since the number of VPNs is a power of two, the entire index space is utilized. However, in many cases if no additional measures are taken then the utilization of the translation table could be quite low, or at worst it may even be impossible to capture all the translation entries because of the table limited size. Such an example is depicted in Figure 3.5. For these eight VPNs, seven LSBs are needed to differentiate them and use them as an index. Therefore, if the outlined above translation technique is to be used in the same manner, this set of VPNs would occupy a memory array with 2^7 entries, while only eight of them will actually be used.

The example outlined in Figure 3.5 therefor shows that a low memory utilization is possible with a large waste of memory and its associated power, if the set of VPNs accessed by the application is targeted as a whole. However, it can be seen that VPNs $0, 1, 4, 5$ can be differentiated by two bits, while the VPNs $2, 3, 6, 7$ can be differentiated by two bits as well. Consequently, if the initial set of eight VPNs is divided into two partitions, $0, 1, 4, 5$ and $2, 3, 6, 7$, then the two LSBs can still be used to form an index into two non-overlapping segments within a translation table as long as information regarding which partition is being used is known prior to access the table. Additionally, the two partitions need to be allocated into two

different four-entry sections of the translation table, so that VPNs across different partitions are guaranteed not to overlap. It can be observed that an alternative partitioning of the VPNs exists as well, which consists of two VPN partitions: $0, 1, 6, 7$, and $2, 3, 4, 5$. This alternative partitioning has the same cost as the aforementioned partition since both pair of partitions require two LSBs for VPN differentiation.

An important constraint for such a partitioning scheme to work properly is that if a load/store instruction can potentially access across multiple VPNs, all these VPNs must be allocated into the same partition. This constraint is needed since because of hardware cost considerations a VPN partition would be identified on a per load/store instruction basis, i.e. a load/store instruction would be mapped to one and only one VPN partition. This restriction guarantees that there is no ambiguity when accessing the DTT. As is shown later in the section, the algorithm forming the partitions is based on this constraint and efficiently finds large partitions with very high index space utilization. Any load/store instruction that cannot be ascertained to access VPNs from the same partition will be handled by a "default" traditional TLB.

The introduced approach involves the identification of partitions of VPNs which result in optimal indexing scheme maximizing the utilization inside each partition while reducing the overall number of partitions. Minimizing the number of partitions, while maintaining high utilization of the translation table is important to control the cost of hardware needed to identify partitions and to compute their translation table index.

After identifying the partitions, each partition is mapped to its own segment

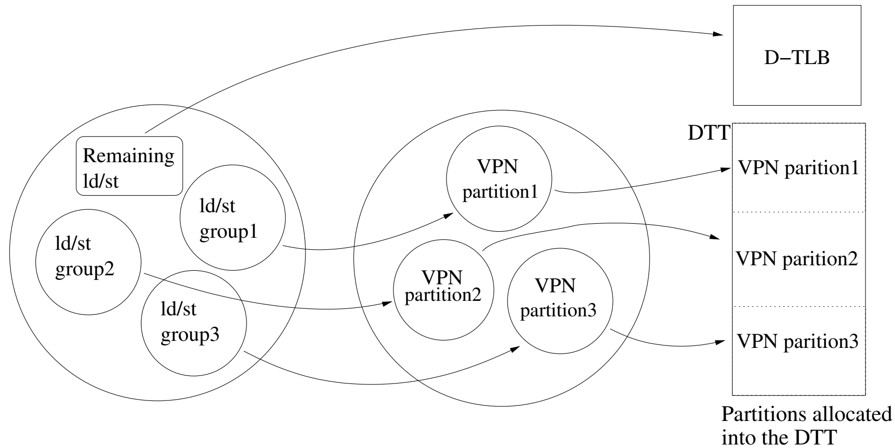


Figure 3.6: Mapping VPN partitions to DTT segments

in the introduced *Direct Translation Table (DTT)*, as illustrated in Figure 3.6. In a manner analogous to scratchpad memories, the DTT is implemented as a small SRAM array, containing the translation entries for all VPNs, which have been determined to be part of VPN partitions. As each partition of VPNs is mapped to a distinct part of the DTT, a special indexing logic is need to form the final index. If the partitions is aligned inside the DTT on address boundaries proportional to their size, a very simple logic is needed to compute the DTT index; the DTT segment offset needs to be simply concatenated to the few VPN LSBs selected as a partition index.

3.1.3 Compiler and OS Support for DTT Management

The role of the compiler support is to determine an efficient VPN partitioning, which maximizes the DTT utilization, and to associate the load/store instructions to VPN partitions as explained in the previous section. The OS support required by

the proposed technique does not go beyond the traditional virtual memory support in terms of maintaining a page table that contains all the address translation entries for the application program. A small additional OS support is required to handle the context switch between tasks, which utilize the DTT. This support is outlined in subsection 3.1.3.3 the multitasking support for DTT management is discussed.

At the heart of the required compiler support is the VPN-partition formation algorithm. This algorithm requires information regarding the statically allocated memory objects of the program together with some profile information. Practically any program spends most of its execution time in a few small parts of its code, which is refered as hotspots. The proposed VPN-partitioning approach is applied independently on each application hot-spot. The application hot-spots are identified through profiling. The partitioning algorithm requires information regarding the application static virtual memory layout for each application hotspot and the access frequencies to the various virtual pages within each hotspot. The first piece of information is available after the compiler/linker maps the data objects into the virtual address space. Dynamically allocated heap memory is treated specially by the DTT approach as outlined later in Subsection 3.1.3.2. The access frequency of the various static memory objects are obtained by the program profile, which is also used to identify the application hotspots.

3.1.3.1 VPN-Partition Formation Algorithm

For a given set of VPNs, there is a minimal number m of LSBs that differentiate all the VPNs. Such a set of VPNs is refered to as an *m-bit* partition, while m is referred to as a *dimension* of the partition. As we saw in the previous section, m depends on the total number n of VPNs in the set, as well as on their particular values. Clearly, $\lceil \log_2 n \rceil$ is a lower bound for m, as this is the minimal number of bits needed to distinguish a set of n elements. If the partition dimension m is equal to $\lceil \log_2 n \rceil$, where n is the number of VPNs, such partition is referred to as *m-bit complete* or just *complete* partition. Consequently, a complete partition is a partition with minimal number of index bits and high utilization of the index space. Such a partition requires a minimal segment of translation table to map its VPNs.

In order to achieve efficient hardware support, we need to identify the minimal number of VPN partitions with the highest utilization of translation table space, subject to the constraint that VPNs accesses by a load/store instruction are placed in one partition. The proposed algorithm partitions the set of VPNs into a minimal number of complete partitions, thus achieving very high utilization of the translation table resources. Consequently, an algorithm is required to find the minimal number of complete VPN partitions for a given set of VPNs. As discussed later, the case of dynamic data allocation and stack memory can also be efficiently dealt with a special partition that is reserved beforehand. For instance, the consecutive set of VPNs where the program stack is allocated would form a complete VPN partition and all the stack references will be associated with that partition.

A first step in the proposed algorithm is to separate the groups of consecutive VPNs. Such groups of VPNs typically correspond to application arrays, buffers, or any data structures, which are allocated into a compact set of consecutive VPNs. All of the static data structures have this property. The program stack can also be though of as exhibiting this property as the stack is always allocated in contiguous region of the virtual address space. As observed in the experimental studies, in many embedded applications these types of data structures constitute an overwhelmingly large part of the application hotspots. Dynamically allocated data structures, which change their memory map are very rare in embedded programs especially in the application hotspots due to their cost in term of performance and power overheads. Furthermore, these initial groups of consecutive VPNs have the very desirable property to constitute *complete* VPN partitions. This can be easily observed from the fact that a set of n consecutive numbers can always be differentiated through the $\lceil \log_2 n \rceil$ LSBs.

Separating the set of VPNs into partitions of consecutive VPNs is only the first step in achieving the desired final result. Very often, an *m-bit complete* partition does not utilize the index range and can, thus, be merged with some smaller VPN partition without increasing the number of least significant bits m to form an index. Figure 3.7 illustrates such a case. In this example, the algorithm starts with three partitions. All of them are complete, as they contain consecutive VPNs; the largest partition, $P1$, is a *4-bit* partition, the next one, $P2$, is a *3-bit*, while the smallest one, $P3$ is a *1-bit* partition. It can be easily seen that the *4-bit* partition can "absorb" the *3-bit* partition, without increasing its initial dimension of 4. The resulting partition

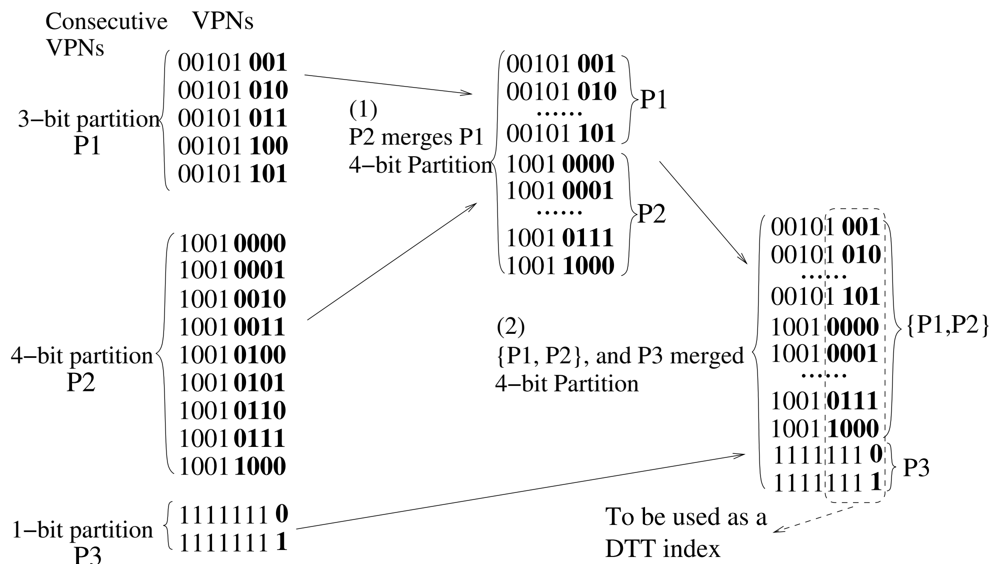


Figure 3.7: VPN partition merging

consists of 14 VPNs and is still a complete partition of dimension 4, yet with much higher utilization of the index space compared to the initial two partitions. In a subsequent step, the *1-bit* partition is merged with the new *4-bit* partition, resulting to a single complete partition of dimension 4, consisting of 16 VPNs, hence having the maximal possible utilization for a *4-bit* partition. The merging steps continue until no such situations as presented in the example exist.

This step in the algorithm fundamentally tries to "pack" the initial set of VPN partitions. At this step, the algorithm needs to follow a strategy for merging the initial VPN partitions in such a way so that minimal number of complete partitions remain at the end with the total partition utilization maximized as well. An important requirement that needs to be imposed here is that at no step of the merging process should the dimension of a partition be increased. That is, when merging two partitions, the resulting partition must have the dimension of the larger parti-

tion from the initial pair of partitions. This constrains ensures that the partition utilization is never decreased in the process of minimizing the number of complete partitions.

In order to merge two partitions, the LSBs of the smaller partition when extended to the dimension of the larger one must not conflict with the LSBs of the larger partition; such conflicts can exist between some of the partitions which would prevent their merging. Therefore, the algorithm to identify the optimal scheme of merging is a multidimensional combinatorial optimization problem, very similar to the well known Bin-Packing problem. An heuristic algorithm, similar to the First-Fit Decreasing [96] heuristic used for Bin-Packing. is developed.

Fundamentally, the aforementioned merging step of the algorithm tries to utilize the empty index space that is left in some partitions. As can be seen from the example in Figure 3.7, partition P2 has 9 VPNs. Since it is a 4-bit partition, the index space of the 4 LSBs is 16, which results in utilization of 9/16 of the available index space and, thus, translation table resources. After the merging steps, though, it can be seen that the utilization of the final partition is 100% as it is a 4-bit partition with 16 VPNs. This final VPN partition can be mapped into a translation table with 16 entries indexed by the 4 LSBs of the VPNs. Consequently, the driving force behind the proposed algorithm is the goal of using the empty space in the initial partitions by fitting there smaller partitions.

Consequently, the merging phase of the algorithm starts with the partition having the largest dimension and then tries to merge as many smaller partitions as possible; the step is repeated until no additional merges are possible. If there

are more than one partition of the same dimension, we use their empty index space as a tie-breaker by picking the one with larger empty space as this increases the likelihood of merging more smaller partitions into the selected one. This step is repeated until all the partitions not merged yet are tried.

The pseudo-code shown in Figure 3.8 describes the proposed algorithm. Step 1 represents the initial formation of complete partitions which correspond to consecutive VPNs. Step 2 is a preparatory step executed prior to the merging phase of the algorithm. Here the initial set of complete partitions is sorted in decreasing order of their dimension. This is needed for the subsequent merging step as the partitions with highest dimensions are first explored as candidates to merge other smaller partitions. Step 3 represents the merging phase of the algorithm. Starting from the partition of highest dimension, all remaining partitions are tried to be merged into the selected one, thus significantly increasing the utilization of that partition without increasing its dimension. Additionally, priority is given to smaller partitions which contain VPNs access by a load/store instruction, which also accesses VPNs from the selected "absorbing" partition. This would improve the likelyhood that if there are load/store instructions accessing VPNs from different initial partitions, these partitions will be merged into a single partition this enabling the utilization of the proposed DTT translation mechanism. Step 4 of the algorithm checks whether there are load/store instructions which can access VPNs from more than one partition. These load/store instructions, if any at all, will be directed to the default D-TLB for address translation. The final step of the algorithm is to select the partitions that can be allocated to the DTT, given the limited DTT size and other hardware limita-

Algorithm input: Set of VPNs

Step 1: Partition the set of VPNs=$\{V_1,\ldots,V_m\}$,

into groups of consecutive VPNs

P=$\{P_1,\ldots,P_n\}$

Step 2: Order the partitioned set P in decreasing order of dimension,

(use number of VPNs and/or access frequency as a tie-breaker);

P=$\{\ldots,P_i,P_j,\ldots,\}$ where $P_i > P_j$

Step 3: For all P_i in P {

For all $P_j <= P_i$ {

if(P_i,P_j are mergeable) {

merge P_j into P_i's empty space

remove P_j

} } }

Step 4: Check every load/store instruction,

make sure it only accesses one partition,

otherwise mark it for the default D-TLB

Step 5: Select the most beneficial VPN partitions

for allocation into the DTT

Figure 3.8: Pseudo-code of the VPN-partition formation algorithm

tions. Because of hardware limitations of mapping load/store instructions to VPN partitions, only a limited number of VPN partitions can be handled by the proposed approach (up to 8, which for all the embedded benchmarks we have experimented with covers all the available VPN partitions). Furthermore, the limited DTT size imposes an additional constraint as well. At this final stage of the algorithm, the VPN partitions which result in most energy benefits are mapped to the DTT. Each partition is evaluated by the execution frequencies of the load/store instructions mapped to them. Subsequently, the most frequently accesses VPN partitions are selected until the DTT space is exhausted. It is noteworthy that this last step of the algorithm is in essence the well-known knapsack problem [97] with each VPN partition having its value in terms of frequency of utilization and also it has its cost in terms of space needed in the DTT. However, for our problem it is the case that the largest partitions are typically the ones that are most heavily accessed and that in the majority of cases all the VPN partitions can be allocated into the DTT. Consequently, a greedy solution at this step provides an optimal solution for any practical purposes.

In terms of algorithmic complexity, it can be noted that the worst-case running the time of the proposed algorithm is $O(n^2)$, where n is the number of partitions in the initial set P of consecutive VPN groups. The worst-case of $O(n^2)$ corresponds to the Step 3 of the algorithm, which is somewhat similar in its traverse pattern to the insertion-sort algorithms. Steps 1, 4, and 5 exhibit linear time complexity in terms of n, while Step 2 can be be implemented with $O(n * log(n))$ time complexity if sorting algorithm such as Heap-sort is used. Therefore, the worst-case running

time of the partition forming algorithm is $O(n^2)$, with n equal to the number of initial partitions. As reported in our experimental results, this number is rather small and for the selected embedded applications always less than 20. This implies that the proposed algorithm imposes no run-time overheads in practice for any embedded software developmen toolchain. As discussed later in the section, the run-time software overhead, which is needed to handle to allocation of DTT is similarly small; it only needs to load the DTT once per application hotspot execution.

At the end of this algorithm, a small set of VPN partitions is produced with very high utilization of the index space. The high utilization implies that very few entries in the DTT segments allocated for these partitions will remain unused. An important requirement in the above algorithm is that in the process of merging, the dimension of the large partition is left unchanged. This might leave a few very small partitions each having several VPNs. However, SRAM arrays are typically implemented as a set of smaller banks. Therefore, as an optional step to the proposed algorithm, it may be beneficial to combine the small partitions into a larger partition with a dimension identical to the memory bank size. This step is performed by starting from the smallest partitions (sizes 1 or 2) and trying to merge them with the next larger one by possibly increasing the partition dimension while keeping it under the memory bank size.

As we mentioned in the previous section, using the DTT for a load/store instruction is only possible if the load/store can generate virtual addresses that belong to the same VPN partition. It is typical that a load/store instruction generates addresses within the same VPN or across adjacent VPNs, especially when it ac-

cesses data structures confined within a region of the address space. Because of the nature of our partitioning algorithm, all such load/store instructions will generate addresses within the same VPN partition. This property ensures that there is a one-to-one mapping between a memory reference instruction and a VPN partition. Consequently, it is at the level of load/store instructions where we identify the VPN segment which is to be accessed. As shown in Section 5.6, the proposed algorithm results in a very few VPN partitions within each application hotspot (fewer than 8), thus, enabling an efficient hardware scheme for mapping load/store instructions to VPN partitions. One such possibility is to use 2 or 3 extra bits from the instruction encoding to mark which VPN partition should be accessed for the particular load/store instructions. The memory reference instructions outside hotspots or the very few load/store instructions which access virtual addresses from different VPN partitions are handled by the default D-TLB.

3.1.3.2 Stack and Heap Memory

Stack memory is typically used to allocate local data of functions and also to preserve various system-level pieces of information such as return address, content of windowed register files, etc. Unless the program employs a deep recursion that depends on the data set, it is relatively easy to statically identify the maximum amount, or at least find a good upper boundary, of stack memory that the program will require while executing. As stack usually grows in consecutive address in the virtual address space, it becomes easy to accommodate stack memory. The set

of consecutive VPNs, which have been identified for stack usage, are entered as a complete partition in the first phase of our VPN-partition forming algorithm. This stack VPN-segment has the same properties as other segments of consecutive VPNs and does not require any special attention from this point on.

Dynamic memory allocation, a software technique rarely used in memory constrained embedded applications, presents an issue that needs special consideration. If an application requires dynamic memory allocation, such allocation can occur inside or outside the application hotspots. It is the more frequent case that such an allocation is typically performed outside the hotspots and only references to these locations are performed inside the hotspots. This follows from the fact that the hotspots are highly optimized parts of the program, where dynamic memory allocation and deallocation is avoided due to its high performance cost. For instance, in nine out of the ten benchmarks that we have considered in our experimental study, no dynamic memory allocation or deallocation occurs inside the applications hotspots. In these nine applications if dynamic memory is used at all, it is always allocated prior to entering the computationally heavy parts of the code. In the benchmark *susan*, a dynamic memory allocation occurs within the hotspot; however, that dynamic memory footprint is very small (a few pages) an can be easily handled as a separate partition in the following way.

Although the virtual addresses for such memory references are not available at compile time, the data heap is normally assigned by the OS and spans consecutive virtual pages. Therefore, a separate VPN partition can be reserved in the DTT, and all the references to dynamically allocated memory are directed to this partition. In

order to establish such a VPN-partition, it is important that an upper bound is found for the dynamically allocated data which is to be accessed within the application hotspot. For the cases where such dynamic memory is allocated outside the hotspots and only accessed inside, it is easy to determine the size of this memory (or an upper bound) either through static analysis or profiling in a way similar to the stack memory. Since the heap region is always assigned to a continuous set of virtual pages, i.e. it is mapped into a well-known and defined region from the virtual address space, these pages can be easily treated as a partition comprising of contiguous VPNs. As we have shown earlier in this section, such complete VPN-partitions can be easily resolved through the $\lceil \log_2 n \rceil$ least significant bits. If the span of the heap allocated objects to be accessed inside the hotspot is too large and cannot be accommodated within the DTT, the references to the heap will be handled by the default D-TLB.

As the physical memory is allocated by the OS at run time, the setup code executed prior to entering a hotspot will acquire the correct physical addresses for the heap VPNs and will store the correct address translation information into the DTT. Inside the hotspot, load/stores to dynamically allocated data are therefore treated as any other memory references and mapped to their own DTT partition, which corresponds to the heap VPNs partition. For the rare cases where dynamic memory allocation is performed inside the hotspot, it is the responsibility of the OS memory manager to update the DTT translation entry in the case of VPNs being mapped to new PPN - this situation is identical to the baseline case where the memory manager updates the TLB in the case of page remappings due to allocation

and deallocation. Additionally, load/store instructions that correspond to pointer accesses, which address can change dynamically while executing the hotspot and thus is not possible to statically map them to any VPN partition, are marked for address translation through the default D-TLB.

3.1.3.3 DTT Management and Multi-Tasking Environments

As this approach can be used in multitasking environments or in a program with multiple hotspots, special care needs to be taken to efficiently utilize the DTT space. If all the VPN partitions of the application can be accommodated within the DTT, they are loaded during system setup. However, when this is not possible due to large number of VPN partitions across all the hotspots, the PIT and DTT have to be initialized with the appropriate data just prior to entering the hotspot, thus sharing DTT space with other hotspots and processes. For this, the compiler is responsible for inserting a special setup code prior to entering the application hotspot. During run-time when entering the hotspot, the special setup code is executed and loads the DDT and the PIT entries with the address translation information regarding the upcoming hotspot.

In order to avoid security shortfalls that arise when the application program is allowed to write any values in the DTT, the setup code can be in the form of invoking a function, which runs within the kernel and guarantees that the actual PPN corresponding to the application VPN segments are loaded into the DTT. Allowing an application code to directly enter its PPNs into the translation hardware

can create possibilities for control hijacking where a buffer overflow can be exploited to map physical memory into the program address space (without OS intervention), which contains malicious code. This can be easily resolved by allowing only the OS kernel to modify the DTT. The setup code inserted by the compiler prior to the hotspots would simply request from the OS that the PPNs corresponding to the particular VPN-partitions used in the hotspot are loaded into the DTT. The application setup code provides the the OS kernel the set of VPN-partitions that it requires for the subsequent hotspot execution. The OS then appropriately loads the corresponding PPNs into the DTT and also remembers this set of VPN-partitions. This set of partitions may be needed by the OS task switch handler as explained below in order to load the DTT for that task again in case it is preempted by another task and later on resumed. In this way if the OS kernel is trusted, then the DTT will be loaded only with valid PPN that corresponds to the application VPNs and are known and established by the OS memory manager.

The performance overhead of this setup code is practically zero as it will take tens or at worst a few hundred cycles, which are to be followed by executing the subsequent application hotspot which usually takes millions of cycles. If a preemptive context switch is needed during the hotspot execution, the DTT and the PIT contents are to be treated as a part of the program's state and need to preserved. Since the DTT content depends on the set of active VPN-partitions, which the application uses when it has been interrupted, there is no need to save the DTT content on task switch. In this case, it is the responsibility of the OS context switch mechanism to only load into the DTT the correct PPNs for the active VPN-

partitions of the preempting task by overwriting the DTT. When the preempted task is resumed, the OS again loads the DTT with the set of VPN-partitions captured as part of the interrupted task state. Fundamentally, the PIT, which is rather small, and the list of active VPN-partitions become part of the process state (PC, status registers, etc.) which is maintained and properly loaded by the OS. They can be stored by the OS at memory locations dedicated by the OS kernel for preserving tasks contexts. In order to speed up this process, the OS can easily use the services provided by a hardware *Direct Memory Access (DMA)* controller to rapidly load the PIT and the DTT content while in parallel preserving the other context, such as the register file and control register by software. We have assumed this implementation strategy when experimenting and analyzing the proposed methodology.

In situations where frequent and extremely fast task preemptions are required, such as in reactive systems [98] where environmental events trigger various tasks, an alternative DTT management could be implemented as well. For such situation where the tasks are short and triggered by asynchronous events, reducing the time to load the DTT may be required. For such cases, an on-demand DTT loading scheme can be implemented. To achieve this, each DTT entry needs to be associated with a task identifier. When a DTT entry is accessed, the task *id* stored in it is compared against the id of the current task. A match would indicate that the DTT entry contains a valid translation information. Otherwise, the default D-TLB is looked up to perform a traditional, general-purpose address translation and the resulting translation entry brought in its place in the DTT. Such a default D-TLB translation is invoked only once per DTT entry while executing the task, since the translation

entry obtained through the default TLB mechanism is loaded into the DTT; any subsequent access to that VPN would find the valid translation in the DTT. In effect, this approach would allow multiple tasks to use the DTT without the need to preserve its content on context switch. The DTT entry would be extended with the task *id*, which purpose is to determine the validity of the DTT translation entry in a way similar used in virtually addressed caches. When multiple tasks have their VPN segments only partially overlapped in the DTT, some DTT entries would be preserved in the DTT for the next execution of that task. In this way, the DTT would not have to be stored/loaded during task switch - this increase in task responsiveness is achieved, however, with the cost of task *id* comparison operations each type the DTT is accessed.

3.1.4 Hardware Support

The proposed approach requires a specialized hardware support which purpose is to map load/store instructions to their corresponding VPN partitions, to compute the appropriate DTT index, and to access the DTT in order to obtain the physical page number. The VPN partitions are identified by using two or three extra bits from the instruction encoding of the memory reference instructions in a way similar to that described in [60]. Our experimental results demonstrate that even for the most complex applications such as the *mpeg video encoder* and the *mp3 audio encoder*, the total number of partitions does not exceed 8. In cases, where there is more than 8 partitions, only the 8 VPN partitions with highest access frequency, or the

maximum number of partitions allowed by the DTT size, will be mapped to the DTT. Consequently, the three bits in the instruction encoding will be used to identify a total of 7 partitions to be allocated to the DTT and the remaining eight value is used for load/store instructions directed to the default D-TLB.

The partition information for each VPN partition is stored in a very small table that is efficiently implemented as a register file with four or eight registers, referred to as the *Partition Identification Table (PIT)*. Each VPN partition is assigned a PIT entry. The partition identifying bits, which we discussed above, are used to access one of these registers, which in turn contain information regarding the VPN partition. The number of VPN partitions which are to be supported by the hardware determine the number of PIT entries. In our experimental results we have shown that supporting 8 VPN partitions per hotspot is enough for all practical purposes. Consequently, we model a PIT with 8 entries. In fact, only 7 entries are needed because one of the eight 3-bit combinations used to identify the partition is used to mark the load/store instructions which are to be treated with the default D-TLB. Each VPN partition is completely defined by its partition dimension m and its offset within the DTT. This pair of numbers is stored in the PIT entry for each VPN partition as shown Figure 3.9. A default value of the VPN partition identification bits is used for the load/store instructions that need to be translated through the default D-TLB.

Each VPN partition must be mapped to an exclusive segment within the DTT. As multiple VPN partitions are mapped, an efficient index computation logic is needed. If VPN partitions are allocated in arbitrary positions within the DTT,

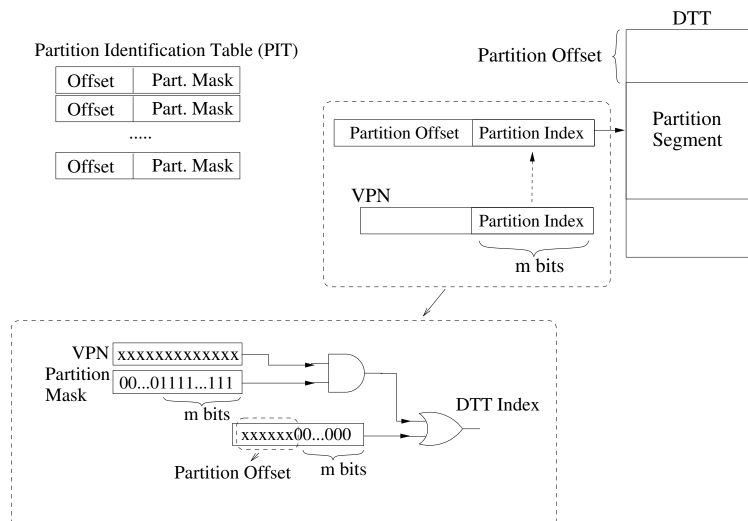


Figure 3.9: DTT index computation and DTT access

accessing such a segment would require the hardware to compute the DTT index by adding the partition offset to the LSB index of that partition. While such an addressing scheme is not difficult to implement, it would introduce a delay on the address translation critical path, possibly increasing the L1 cache access time or the on-chip memory access time. We propose, instead, an alternative hardware scheme, which requires only a concatenation of the partition LSB index and the DTT segment offset. This could be achieved by allocating the VPN partition at an address boundary multiple to the partition size, i.e. align the VPN partition in the DTT by the partition dimension. For instance, a partition of dimension 5 can be allocated at DTT addresses, which are multiple of 32. When such an alignment is implemented, in order to compute the final DTT index, the partition offset and the m LSBs of the VPN are simply concatenated. This scheme and the required hardware logic are illustrated in Figure 3.9. To facilitate the hardware implementation, instead of storing directly the dimension m for each partition, a partition mask is used instead,

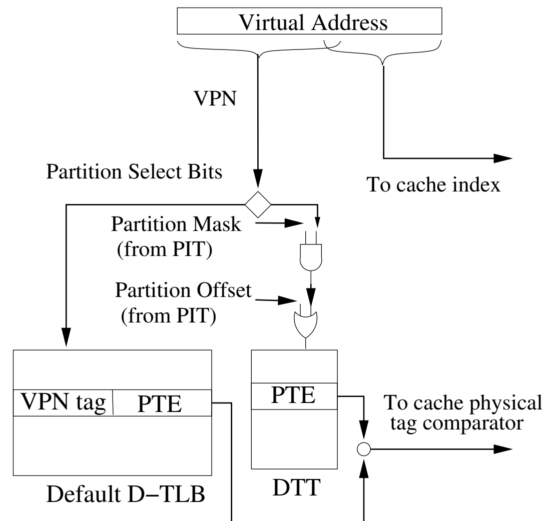


Figure 3.10: Hardware architecture

containing m 1s on the least significant bits, extended with zeroes towards the most significant bits. By using such a representation, the logic needed to compute the final DTT index comprises of one *OR* and one *AND* gate per bit position. The pairs of *Partition Offset* and *Partition Mask* defining each partition are stored in the PIT. As demonstrated in Section 5.6, eight such entries are typically enough to target all the VPN partitions for each hotspot.

Figure 3.10 shows the entire address translation architecture. The load/store instructions mapped to VPN partitions are directed to the DTT and their addresses translated rapidly, energy-efficiently, and in a time-deterministic manner. The remaining few load/store instruction, if any at all, are directed to the default D-TLB for traditional address translation.

It is noteworthy that the lookup into the PIT is performed early in the pipeline when the load/store instruction is decoded. Therefore, the PIT lookup is outside the critical path of cache lookup or data memory access. Only the circuitry for

computing the DTT index, which consists of two logic gates per bit, is needed when the virtual address is generated by the processor. In our experimental results we show the access times for the DTT tables that we experiment with and compare them with the access time of the baseline D-TLB. From these numbers it is evident that the introduced DTT-based address translation does not increase the critical path delay for memory accesses. The area overhead of the proposed approach is also minimal. Even though we are introducing a new SRAM array for the DTT table, the size of the default D-TLB is much smaller than the baseline D-TLB typically used in such systems. This can be done without any sacrifice of performance since very few load/store instructions will utilize the default D-TLB; in our experimental study we report on this as well.

3.1.5 Analysis and Discussion

3.1.5.1 Real-Time Performance Improvements

The proposed address translation organization greatly improves on the worst-case timing analysis for real-time applications. All the load/store instructions allocated to the DTT are guaranteed a single cycle translation with performance implications identical to a guaranteed TLB hit. This situation is very similar to the application of scratchpad memories (SPM) where memory references to data allocated in the SPM are guaranteed a single cycles memory access. As can be seen from our experimental results, the great majority of load/store instructions for all benchmark hotspots are guaranteed translation through the DTT. Such guarantees,

which are known at compile-time will greatly improve the upper bounds of the *Worst Case Execution Timing (WCET)* analysis of the code as for most of the memory reference a single cycle address translation is guaranteed. The static WCET analysis [95, 99] is a traditional compile-time step for any real-time application, which obtains guarantees for the worst-case scenario in executing a particular piece of code. Traditionally, when it cannot be inferred by the WCET analysis that a memory reference will always hit in the TLB, the worst case, i.e. TLB miss, must be assumed. With the proposed DTT methodology, however, the majority of load/store instructions (and in many cases all of them) can be guaranteed a single cycle address translation by our VPN partitioning algorithm and DTT allocation methodology. Consequently, the upper bounds of execution times produced by WCET analysis can be greatly reduced. Only for the very few load/store instructions, which are left for translation by the default D-TLB a worst case timing pertinent to the particular processor architecture needs to be used.

It is noteworthy that the proposed DTT architecture does not guarantee single cycle address translations for any program or program hot-spot. This is due to the fact that for some applications with large memory footprint, it may be possible that not all of the load/store instructions (and VPN partitions) can be allocated to the DTT. Consequently, these few remaining load/store instructions need to be handled by a traditional TLB with possibility for misses and multiple cycle page table traversals. However, in our experimental study we have found out that for many applications or applications hot-spots a reasonably sized DTT of 256 entries covers all the VPN partitions and as such provides guaranteed single cycle address

translation for all memory references. Whether all of the VPN partitions for a given application hot-spot can be covered by the DTT translation mechanism depends both on the application memory footprint and the DTT size.

3.1.5.2 Leakage Power

As the approach we offer fundamentally targets dynamic power consumption, in our experimental study we report only dynamic power reductions. Even though our approach introduces an extra SRAM array to implement the DTT, it must be noted that the default D-TLB is used for the very few references outside the hotspots and the ones, which cannot be handled by the DTT. From our results it can be seen that the majority of memory references (frequently even all the references) inside the hotspots can be handled by the DTT. Consequently, the D-TLB that is needed for the remaining few references can be significantly reduced in size as compared to the baseline architecture with traditional TLB only. Without sacrificing performance and without introducing new translation misses the default D-TLB can be left with only a fraction of the usual number of translation entries, thus saving silicon ares from both the reduced tag and data arrays. This reduction of the tad and data arrays will offset the introduced SRAM array for the DTT and its leakage power. As reported and discussed in the next section, our default D-TLB is limited to 16 entries only and the entire area of the proposed translation hardware is smaller or comparable to a typical baseline TLB of 64 entries. Consequently, we estimate that our approach does not have a practical impact on the leakage power. Furthermore,

as can be seen from our experimental results, for many of the application hotspots the default D-TLB can be entirely turned-off or placed in drowsy mode [100, 101] during the hotspot execution time since all of the memory references inside the hotspots can be handled by the DTT. Consequently, for such hotspots, the D-TLB will consume no leakage power. The small TLB needs to be kept operational only for the very short periods of time (less than 5% of the entire execution time) when the program executes code outside the hotspots.

3.1.6 Experimental Results

In evaluating the proposed techniques, we have performed a quantitative analysis and comparison of baseline D-TLB architectures and the proposed application-driven address translation organization. The baseline D-TLB organizations contain 64 entries, with 4-way and 8-way set associativity (64-4SA, 64-8SA), as well as fully associative (64-FA). The virtual page size is conventionally fixed to 4K. We have evaluated two DTT organizations, one with a DTT size of 256 entries and another one with 128 entries. The energy per access, time delay, and area of the baseline D-TLB organizations as well as the DTT structures have been obtained by using the CACTI-3 tool [102] for a 0.18μ process technology. As a default D-TLB in the DTT configurations, we have utilized a 16-entry 2-way set associative (16-2SA) and a fully associative (16-FA) TLB structures. Because of the significant difference in terms of energy and area between set associative and fully associative TLBs, we have evaluated a DTT configuration with fully associative default TLB to compare

	Energy (nJ)	Delay (ns)	Area (cm2)
64-4SA	0.405366	1.037	0.001732
64-8SA	0.71175	1.08001	0.002108
64-FA	0.176706	1.80768	0.002270
16-2SA	0.238378	0.96902	0.000738
16-FA	0.105995	1.58034	0.000934
256 DTT	0.114782	0.795045	0.001904
128 DTT	0.103756	0.743074	0.000846

Table 3.1: TLB and DTT characteristics

it against the 64 entries fully-associative baseline configuration.

Table 3.1 shows the energy, time, and area characteristics of the evaluated TLB and DTT structures. From this table it can be seen that the DTT configuration of (128-DTT + 16-2SA D-TLB) occupies less area than any of the 64-entry baseline TLBs, while the configuration of (128-DTT + 16-FA D-TLB) is comparable area-wise to 64-4SA and is smaller than the 64-8SA and the 64-FA baseline TLBs. It can also be seen from these data that the proposed DTT introduces no performance overhead, as its delay is smaller than the delay of the baseline TLBs. The access to the PIT is performed early in the pipeline, when the load/store instruction is decoded, hence it does not contribute to the DTT delay. Another interesting observation from the data in Table 3.1 is that the per access energy to the fully-associative TLB

is smaller than the per access energy to the same sized but set associative TLBs. However, the delay and area characteristics of fully associative TLBs are worse than the set associative organizations. For this reason, we have used as a default D-TLB in the DTT configuration both set-associative and fully associative organization in order to produce consistent comparisons with the baseline TLBs.

	h-sp	Freq(%)	Enrg 64-4SA	Misses 64-4SA	Enrg 64-8SA	Misses 64-8SA	Enrg 64-FA	Misses 64-FA
adpc	1	100	0.21	3	0.37	3	0.09	3
g721	1	100	14.3	2	25.1	2	6.22	2
gsm	1	100	20.9	3	36.6	3	9.09	3
epic	1	100	2.77	2295	4.86	2323	1.21	2354
jpeg	2	11,72	2.38	53	4.19	53	1.04	53
mpeg	3	81,1,11	138.5	3026	243.1	3046	60.4	3122
mp3	5	25,13,24 16,18	128	53316	224.8	60360	55.8	61969
susan	1	100	0.58	17	0.33	17	0.15	17
sha	1	100	2	8	1.14	8	0.5	8
aes	1	100	60.7	5	34.6	5	15.1	5

Table 3.2: Baseline D-TLB characteristics and energy (in mJ)

Table 3.2 shows the baseline characteristics. The first column in the table contains the benchmark name. The first 6 applications are from the Mediabench [103] set of benchmarks, while the seventh application is a widely used open source *mp3* encoder (http://www.mp3dev.org/). The last three benchmarks are from the MiBench [104] collection. The 10 benchmarks we have used in our study cover the important domains of speech, audio, image, and video processing, as well as two important encryption tasks (*sha*, and *aes*). The subsequent column shows the number of hotspots identified for each benchmark with the execution frequency for each hotspot in percentage presented in the next column. The application hotspots have been identified through profiling and simulation. The next three pairs of columns show the energy consumption (in μJ) and the number of TLB misses for the three baselines TLB organizations.

A banking memory architecture is assumed with each bank having 32 DTT entries. Consequently, partitions are merged to maximum dimension of 8 and minimum of 5. Furthermore, if the VPN partitions of the application do not occupy the entire DTT, the unused DTT banks are turned off. The maximum number of partitions per hotspot is set to 7, thus resulting to a total of 7 PIT entries, each consisting of 8-bit *partition offset* and 8-bit *partition mask* (7-bit for the 128 entry DTT). From the results reported in Table 3.3, it can be seen that the number of VPN partitions per hotspot is always below eight even for the most complex benchmarks. Table 3.3 reports all the information regarding the VPN partitions that is independent from the DTT size. The first column (I.Part.) shows the number of initial VPN partitions for all the benchmark hotspots. This number corresponds to

	I. Part.	Part.	Dim.	#VPN	Util.(%)	ld/st	Freq.
adpc	2	1	2	3	75	11	100
g721	2	1	2	2	50	103	100
gsm	3	1	2	3	75	772	100
epic	4	3	1,7,7	2,74,87	100,58,70	951,547,34	9,32,59
jpeg	4	3	1,3,5	1,4,32	50,50,100	228,77,24	65,14,21
	7	1	6	44	69	1709	100
mpeg	5	3	1,6,7	1,55,66	50,86,52	606,204,42	11,13,76
	3	2	1,7	1,77	50,60	6,33	1,99
	5	4	1,5,7,7	2,25,88,99	100,78,69,77	27,30,2,7	53,30,2,15
mp3	8	4	1,4,5,5	1,15,17,27	50,94,53,84	1,349,104,770	1,6,31,62
	11	2	5,5	16,26	50,81	961,458	87,13
	7	3	1,3,4	2,5,8	50,63,50	75,65,22	29,3,68
	7	2	1,4	2,11	100,69	10,439	2,98
	12	3	2,4,5	3,11,19	75,69,59	22,1149,156	3,82,15
susan	5	2	1,4	2,15	100,94	104,576	1,99
sha	4	2	2,3	3,5	75,63	204,142	86,14
aes	2	2	2,1	4,1	100,50	224,151	53,47

Table 3.3: VPN-Partition Information

the number of partitions produced by the first step of our partition forming algorithm. The next column (Part.) presents the number of VPN partitions for each hotspot after the termination of the proposed algorithm. It can be observed that the algorithm merges a significant number of the initial groups of consecutive partitions and thus produces a relatively small number of compact VPN partitions. For instance, the *jpeg* benchmark has two hotspots, where the first one exhibits three VPN partitions, while the second hotspot has only one partition. The next column, labeled with *Dim.* shows the dimension for each partition. The three partitions of the first hotspot of *jpeg* have dimensions of 1, 3, and 5 respectively, while the single partitions of the second hotspot is of dimension 6. As described earlier in the section, the partition dimension is equal to the number of least significant bits from the VPN, which will be used to access the DTT. The next column, labeled with *#VPN* reports the number of VPNs for each partition. The format for this information is similar to the previous column. The next column shows the utilization of each partition in percentage. Utilization is defined as the ratio between the number of VPNs and $2^{Dim.}$, i.e. how much of the index space has been used for actual VPNs. It can be observed that our algorithm consistently achieves utilization ratios at and above 50%. The subsequent column shows the number of load/store instructions, which have been associated with each VPN partition. As explained earlier, each load/store instruction is either mapped to a single VPN partition and translated through the DTT or routed for address translation to the default D-TLB. The last column in Table 3.3 shows the access frequency for each VPN partition. This data is the ratio between the number of accesses to VPNs in the partition and the total

number of memory references for the hotspot. From the data in this table it can be concluded that the proposed approach forms a relatively small number of VPN partition with high utilization of the index space.

The simulation is performed with the SimpleScalar toolset [105]. Optimization level -O2 has been used when compiling the benchmarks with the provided *gcc 2.7.2* cross-compiler. Since the proposed approach exploits knowledge regarding the virtual address space mapping of the program data objects and not the particular order or pattern in which they are accessed, we don't expect any major deviations in the reported results if more aggressive compiler optimizations are used. Through benchmark simulation and analysis, the hotspots and their virtual memory layout are identified. For the proposed address translation technique, DTTs of 256 and 128 entries are evaluated, where each DTT entry consists of four bytes. The number of the *AND* and the *OR* gates for computing the DTT index is set to 8 due to the DTT size. Additionally, a default D-TLB with 64 entries is assumed for VPNs outside the hotspots, and for VPNs inside hotspots but not included in directly translated VPN partition. The DTT access energy is obtained by using CACTI; this is achieved by subtracting the tag-related energy from the total energy of a direct mapped cache. The access energy of the PIT register file is estimated by using the data presented in [106]. The energy for 0.2μ and 2V Vdd process technology parameters has been scaled down to 0.18μ, 1.7V Vdd process technology by applying the same estimation methodology as the one utilized in CACTI. The power consumption of the few logic gates needed in the computation of the final DTT index is accounted for as well, even though their contribution is orders of magnitude less than the power consumption of

the DTT table. After applying the proposed VPN partition generating algorithm, a setup code is inserted on the entrance to each hotspot; the so instrumented program is subsequently simulated with a modified version of the SimpleScalar where we model the impact of the proposed address translation architecture. This setup code has no impact on the total power consumption for all practical purposes, as it is executed only once on the entrance of the application hotspots, which typically execute for tens of millions of cycles. The final power consumption is computed by summing up the energy for all the VPN to PPN translations including the energy needed for all the hardware we have introduced. The reported data accounts for all the memory references, including the ones to dynamically allocated (heap) and stack memory. During the short periods of time when the program executes outside the hotspots, we have accounted for the traditional TLB address translations. Additionally, we show the total number of address translation misses for both the baseline and the proposed architectures.

Table 3.4 shows the energy dissipation and reduction for the proposed address translation methodology with a 256-entry DTT and a default D-TLB of 16 entries. The first column shows the DTT utilization for each benchmark and each hotspot. DTT utilization is defined as the ratio in percentage of the actively used DTT entries and the total number (256) of DTT entries. For the *adpcm*, the *g721*, and the *gsm* benchmarks it is rather low (0.78%) because of the very small number of VPNs accessed by these applications. The next column shows the number of VPN partitions allocated to the DTT. It is shown how many of all the available partitions for each hotspots have been mapped to the DTT. The number of covered

	DTT util.	Part Cover.	DTT Freq.	Enrg. 16-2	Red. 64-8	Red. 64-4	Misses 16-2	Enrg. 16-fa	Red. 64-fa	Misses 16-fa
adpc	1.56	1/1	100	0.06	83.9	71.7	0	0.06	35.0	0
g721	1.56	1/1	100	4	83.9	71.7	0	4	35.0	0
gsm	1.56	1/1	100	5.9	83.9	71.7	0	5.9	35.0	0
epic	100	2/3	90.9	0.86	82.3	68.9	2	0.78	35.1	2
jpeg	16,25	3/3,1/1	85.9	0.78	81.4	67.4	185	0.67	35.8	32
mpeg	76,51 63	3/3,2/2 3/4	92.9	42.2	82.6	69.5	408102	39	35.4	12270
mp3	32,25 10,7 20	4/4,2/2 3/3,2/2 3/3	97.3	37.3	83.4	70.9	10891	36.2	35.2	11436
susan	7	2/2	100	0.09	83.9	71.7	0	0.09	35.0	0
sha	4.69	2/2	100	0.32	83.9	71.7	0	0.32	35.0	0
aes	2.34	2/2	100	9.78	83.9	71.7	0	9.78	35.0	0

Table 3.4: Direct address translation with 256-DTT

	DTT util.	Part Cover.	DTT Freq.	Enrg. 16-2	Red. 64-8	Red. 64-4	Misses 16-2	Enrg. 16-fa	Red. 64-fa	Misses 16-fa
adpc	3.13	1/1	100	0.05	85.4	74.4	0	0.05	41.3	0
g721	3.13	1/1	100	3.65	85.4	74.4	0	3.65	41.3	0
gsm	3.13	1/1	100	5.34	85.4	74.4	0	5.34	41.3	0
epic	100	1/3	59	1.09	77.7	60.8	8546	0.72	40.8	183
jpeg	33,50	3/3,1/1	85.9	0.72	82.8	69.7	185	0.61	41.1	32
mpeg	100 100,27	1/2 1/2,2/4	71.4	48.6	80	64.5	412858	35.7	40.9	12766
mp3	64,50 20,14 41	4/4,2/2 3/3,2/2 3/3	97.3	33.9	84.9	73.5	10891	32.9	41.3	11436
susan	14	2/2	100	0.08	85.4	74.4	0	0.08	41.3	0
sha	9.38	2/2	100	0.29	85.4	74.4	0	0.29	41.3	0
aes	4.69	2/2	100	8.85	85.4	74.4	0	8.85	41.3	0

Table 3.5: Direct address translation with 128-DTT

partitions depends on the DTT size, the total number, and size of VPN partitions; each partition is allocated, if possible, at DTT offset aligned with the partition dimension. It can be seen, for example, that for the *epic* benchmark, only two of the 3 partitions are mapped to the DTT, while for the *jpeg* all the partitions have been allocated to the DTT. The partition selection processes, which was described in details in Section 3.1.3.1, takes into account the access frequency of each partition and the DTT size. The third column, labeled with *DTT Freq.*, shows the frequency in percents of DTT utilization. This number is the ratio of the address translations handled by the DTT and the total number of address translation including the ones performed through the default D-TLB. The subsequent column shows the energy dissipation (in mJ) for the 256-enrty DTT with a 2-way set associative 16-entry default D-TLB (DTT/16-2). The next two columns show the energy reduction of this DTT configuration compared to the 64-8SA and the 64-4SA baseline TLBs, while the subsequent column displays the total number of the default D-TLB misses. It can be observed that compared to the baseline configurations, a large number of the TLB misses is eliminated. The column, labeled with *Enrg (16-fa)*, shows the energy dissipation (in mJ) for the same DTT but with a fully associative default D-TLB and this energy is compared to the fully-associative baseline (64-fa) and the reduction in percentage is presented in the subsequent column. This is followed by number of misses of the default fully-associative (16-fa) D-TLB misses. Similarly, the number of misses is greatly reduced compared to the baseline fully-associative case.

Table 3.5 shows the same information but for a 128-entry DTT. Again, two

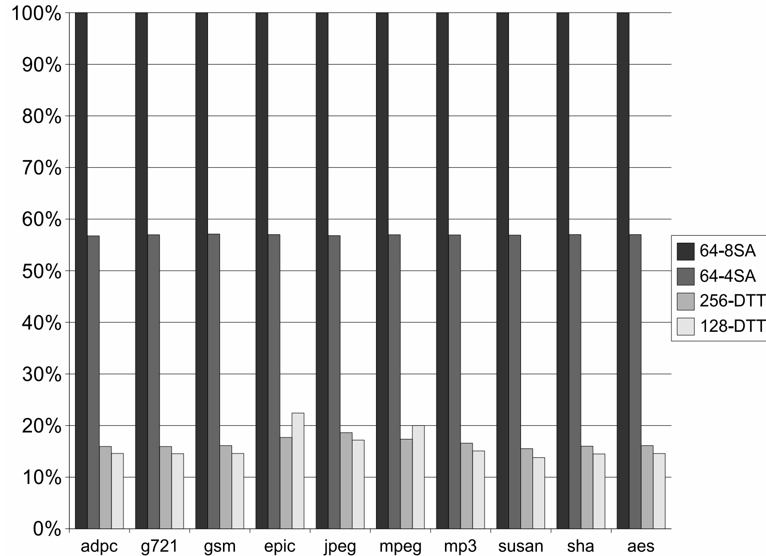


Figure 3.11: Energy comparison (normalized) for set-associative organizations

default D-TLB organizations are explored: 16-2SA and 16-FA. Due to the smaller DTT for a few of the hotspots some partitions that were covered with a 256-entry DTT will not be allocated to the smaller DTT. For instance, it can be seen that for the *epic* benchmark, only one out of the three partitions can be covered, which, of course, results in less energy reductions compared to the 256-entry DTT. This is because even though the smaller DTT consumes less energy, more memory references will be routed to the default D-TLB for address translation, which consumes significantly more power than the DTT table.

Figure 3.11 shows a bar diagram of normalized total energy dissipation for both the 128- and 256-entry DTT case with 16-2SA default D-TLB compared to the set-associative baseline TLBs. The energy consumption for the 64-8SA baseline is normalized to 100%, while the energy for the other baseline (64-4SA) as well as the two DTT cases are scaled accordingly. It can be readily observed that the

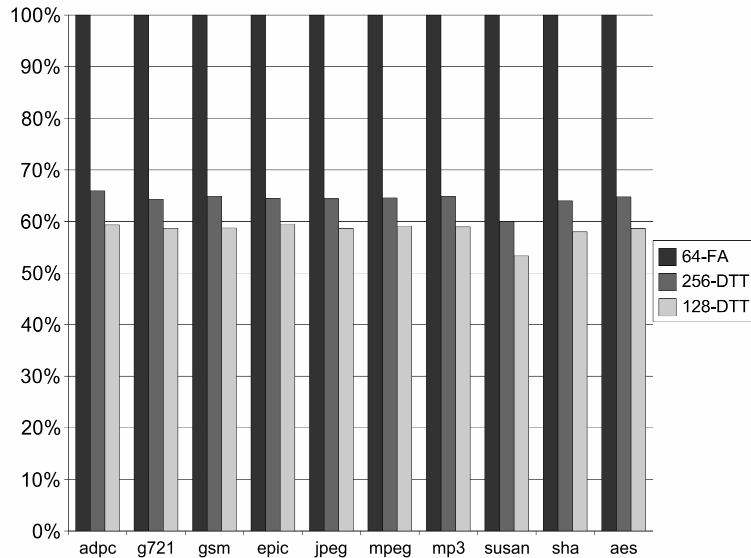


Figure 3.12: Energy comparison (normalized) for fully associative organizations

proposed approach consistently reduces the energy needed for address translation to a 15% - 25% fraction of the energy consumed by the 64-8SA baseline, and to a 20% - 35% fraction of the energy needed by the 64-4SA case. It can be also seen that for the *epic* and the *mpeg* benchmarks, the 128-DTT consumes slightly more energy than the 256-DTT case. This can be explained by the fact that for these two benchmarks, the smaller 128-entry DTT cannot handle all of the VPN partittions that the 256-DTT can translate, thus resulting in more default D-TLB lookups, which are more energy consuming than the DTT. Figure 3.12 shows a similar comparison but for the fully-associative baseline (64-FA) compared to the 256- and to the 128-entry DTTs with 16-FA default D-TLB. For this case, the energy reductions are smaller than the ones achieved for set-associative organizations, because the relatively small fully-associative buffers consume slightly less energy compared to a traditional set-associative implementation with multiple tag arrays. This data

was shown in Table 3.1 as obtained from the latest version of the Cacti tool. However, the small fully-associative buffers exhibit a worse timing delay as compared to set-associative implementations. Consequently, if the default D-TLB is always selected to be 16-FA, then the propagation delay of this organization would be worse than a baseline 64-4SA or 64-8SA TLBs (but better than 64-FA). This is another reason to introduce the two choices of 16-2SA and 16-FA as a default D-TLB, so that depending on the baseline, the appropriate choice of default D-TLB is made in order to stay within the timing of the baseline organization and thus not impact the clock cycle time.

3.2 Heterogeneously Tagged Cache

3.2.1 Introduction

In the presence of virtual memory, caches can be accessed in several different ways. Since there are both virtual and physical addresses present in the system, either one can be used to access the cache. Furthermore, the cache access operation can be split into two components: indexing and tagging. Consequently, four types of cache access mechanisms can be constructed depending on the type of address used for indexing and tagging. If virtual addresses only are used to access the cache, the resulting cache is referred to as *virtually-indexed and virtually-tagged*. The benefits of this cache architecture is that there is no need for address translation when accessing the cache, which results in fast access time and, even more importantly for embedded processors, low power consumption [53, 66, 107]. Virtually indexed

and tagged caches, however, exhibit severe drawbacks namely cache *aliasing* and *synonyms* [108, 107]. Cache aliasing is a situation where the same virtual address from different tasks is mapped to different physical addresses. Such a situation necessitates flushing the cache on context switch, which can lead to significant performance degradation. The aliasing problem is avoided if the tags are extended to store a *Process IDentifier (PID)*, a unique key associated with each process. Additionally, since no TLB lookup is performed, each cache line must be extended to capture the access control bits for the cache line address range. At the same time, however, the cache synonym problem has been traditionally difficult to overcome. Cache synonyms occur when different virtual addresses, usually from different virtual address spaces, are mapped to the same physical address. This situation occurs naturally when two processes share data. If virtual addresses are used to access the cache, the different virtual synonyms of the shared physical location will end up in different cache blocks. This, in turn, can easily lead to cache coherence problems if one of the processes writes into the synonym location, leaving the other cached copies of the shared data stale. Various solutions for avoiding cache synonyms have been offered for general-purpose processors [66, 53] all of which introduce non-trivial hardware structures with significant power overhead, thus making their adoption in embedded processors infeasible. Because synonyms occur when sharing writeable data, virtually indexed and tagged caches have been mostly used as I-caches where such sharing usually does not exist.

Physically-tagged and physically-indexed caches are the exact opposite. In this cache architecture, only physical addresses are used for indexing and tagging. Nat-

urally, aliasing and synonyms are no longer an issue. However, address translation needs to be performed for each cache access and before forming the cache index, which results in delayed cache access time and significant power overhead.

Physically-tagged and virtually-indexed caches are the most typical D-cache architecture for processors with virtual memory support. By performing address translation only for the tags, the cache indexing is overlapped with the tag translation, thus effectively hiding the address translation latency. By imposing certain restriction to the OS memory manager and by introducing additional hardware support, the physically-tagged and virtually-indexed cache eliminates the cache synonym problems with no performance implications. However, the power consumption of such a cache architecture is quite high as address translation is performed each time the cache is accessed in order to obtain the physical tag.

In this section, a novel cache architecture, which *selectively uses either physical or virtual tags* with the objective of minimizing the number of address translations is investigated. In this way the power benefits of virtual caches are combined with the synonym elimination benefits of physically-tagged caches. Application-specific information regarding the type of memory references is used to determine whether to use a virtual or a physical tag. Tag translation is performed only for memory references which can potentially refer to shared memory and as such result in cache synonyms. The majority of cache accesses which refer to private data in the process' virtual address space, i.e. pages only mapped to that address space, are being handled with virtual tags, thus necessitating no address translation on cache access. Not only is the majority of tag related address translations eliminated, but also for

the shared-data memory references we introduce a new translation scheme, which utilizes the knowledge of physical page locations in order to replace the power expensive TLB lookups with simple arithmetic operations. This novel *cache tagging* approach coupled with *specialized address translations* for the shared physical pages results in very energy efficient cache operations.

3.2.2 Heterogeneous Cache Tagging - A Functional Overview

The proposed cache tagging policy takes into account application information regarding the page status of the memory location being accesses. A virtual page, which is known to be mapped to a physical page, which in turn is mapped to at least one other virtual page from another address space (belonging to another process) is considered to be shared. Such mappings are established and controlled by the OS on the request from the application or when there is a need to share code or data between different address spaces. Physical tags are used with the purpose of resolving the serious problems which such shared memory pages can cause when stored in the data cache. The proposed heterogeneous cache tagging in its essence is a hybrid approach which combines the low-power benefits of virtual tags, with the cache synonym avoidance properties of physical tags.

Synonyms are multiple virtual addresses which are being mapped to the same physical address. Synonyms can naturally appear when a shared data in physical memory is being mapped to different virtual address in more than one tasks in order to facilitate data communication between processes. With synonyms, only a single

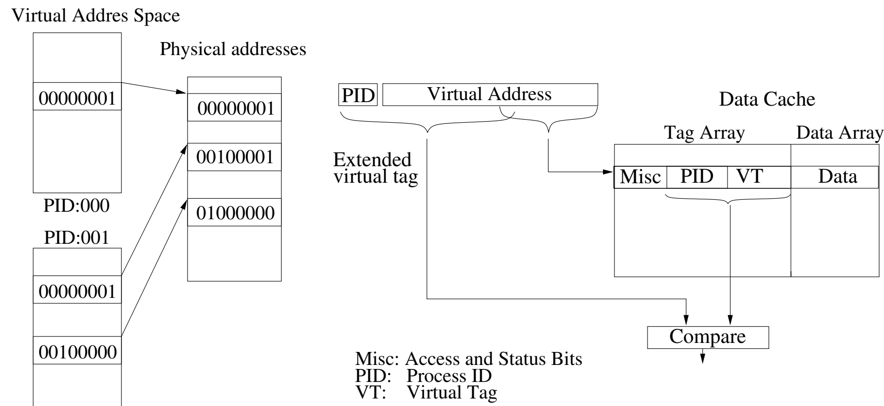


Figure 3.13: Using PID to eliminate aliasing with virtually tagged caches

copy of the shared data is maintained in physical memory, which is very beneficial for embedded systems which typically have limited memory resources. Not only is memory usage minimized, but the performance overhead of copying the shared data between the processes address spaces is eliminated.

In general-purpose processors, the application usually comes in binary-only format. Consequently, no information regarding data sharing is available to the microarchitecture which assumes that a large variety of programs will be executed. Thus every memory access is assumed to be a potential synonym address. Embedded processors and systems, however, have the distinctive advantage of complete application knowledge, as the embedded software is usually developed concurrently with the hardware design or is available in a source code format.

For example, if it is known in advance that there is no shared data mapped to different address spaces or there is no shared data at all in the target embedded system, then it is clear that no cache synonyms could exist. Figure 3.21 shows an example of two processes sharing the cache and no shared physical pages. It can be

seen that the first virtual pages from both process have the same virtual address, but they are mapped to different physical pages. In situations like this when no synonyms are possible, the combination of process ID and virtual address can serve as the single identifier to differentiate among all data addresses. Even though the virtual address of the two first pages from both address space are identical, the *Process IDdentifier (PID)* of each process which extends the tag would suffice to distinguish the two identical virtual addresses. Thus no physical address is needed to locate the data in the cache, and the TLB lookup step prior to access the cache could be avoided completely. Consequently, the cache for such references can be virtually-indexed and virtually-tagged with tags extended with process IDs. Only when physical memory needs to be accessed in the cases of a cache miss or a cache write-back, the virtual address is translated into physical address through the TLB. Additionally, the *Access Control (AC)* bits and other status bits are associated with each cache block and obtained from the TLB when the data block is placed in the cache. For virtually-tagged caches it is inevitable that the cache line status bits need to be extended to contain the access control for the cache line address. Since no TLB lookup is performed on cache access (and hit) the AC bits (usually from 2 to 4) need to be present in the cache.

In general, however, it is common that multiple processes working under the same application would need to share data in order to communicate with one another. While the majority of the data accesses are typically non-shared data which can use virtual tags with the traditional for virtually-tagged caches extension of PID and AC, as shown in the example in Figure 3.21, special care needs to be taken for

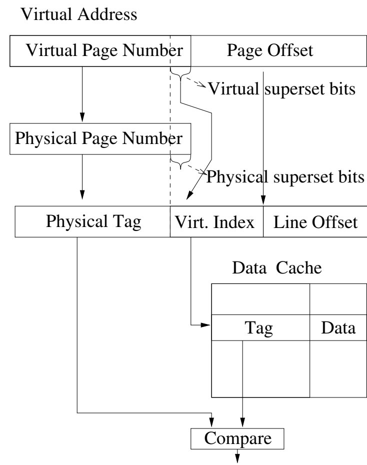


Figure 3.14: Superset bits - overlap between virtual index and VPN

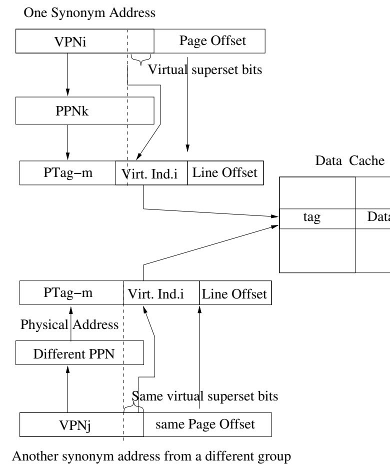


Figure 3.15: Conflicting cache indices for the case of synonym groups exhibiting identical physical tags and virtual superset bits

the memory references mapped to shared physical pages.

A virtual address consists of a VPN and a page offset. When used to access the cache, the same address is split into block offset, cache index field, and a tag. A *synonym group* is the set of VPNs from the different processes, which are mapped to the same physical page. In order for the cache to work properly it must be ensured that all the synonyms from the group are mapped to the same cache location in order to access the shared physical page. If the cache index part of the address is completely contained within the page offset part, thus implying that the cache index from the virtual address is identical to the physical index, then no synonyms can occur if physical tags are used. When the virtual page size, however, is smaller than the cache size divided by the associativity, then a fraction of the most significant bits

of the cache index field overlaps with the VPN. The intersection bits of cache index and VPN are called *superset bits*, or *color bits*. The set of synonyms are *aligned* if their superset bits are identical to the superset bits of the physical address, as shown in Figure 3.14. In this case, the virtual cache index is the same as the physical index, and consequently the physical tag is sufficient to differentiate among synonyms in the cache. If the superset bits are not identical and do not match with the corresponding bits from the physical address, which is often the case when no special care is taken, it becomes possible that the same location in physical memory is cached at two different cache locations. Furthermore, if the synonyms from the same groups are only aligned in the virtual address space, i.e. the virtual superset bits are identical but do not match the physical superset bits, then it is possible that this synonym group may conflict in the cache with another memory location which happens to have the same virtual index, virtual superset bits, and physical tags, but different physical superset bits; such a conflict will not be resolved in the cache with the physical tags as they are identical. This situation is illustrated in Figure 3.15. Of course, this situation cannot happen if the synonym group is completely aligned (both in virtual and physical address space) or larger physical tags are used that overlap the superset bits.

Traditionally, general-purpose processors with virtual memory utilize virtually-indexed and physically-tagged caches. To avoid the synonym problem, the OS memory manager is required to align the set of all synonyms to the physical page frames to which they map, i.e. provide for a complete superset alignment in both virtual and physical address spaces. Such a requirement imposes a significant constraint

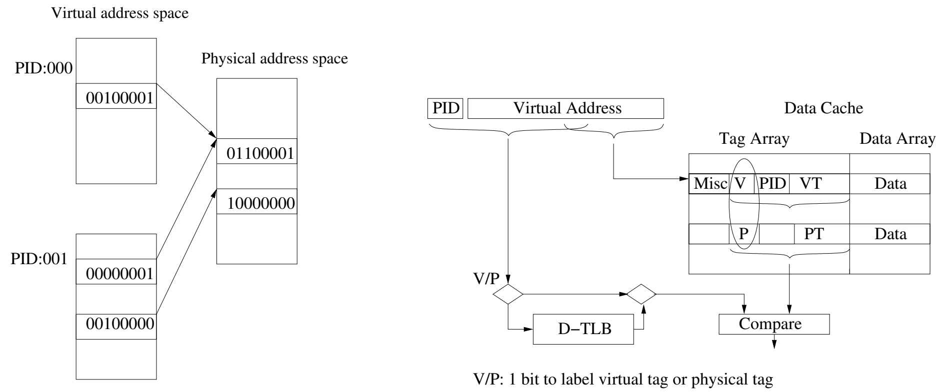


Figure 3.16: Two processes sharing one data block; aligned virtual addresses

in terms of physical memory utilization, as physical frames can be placed only in a subset of all possible page locations. This could impact the page fault rate in general-purpose systems, while in the case of embedded system with limited physical memory it could result in cases where such alignment is simply not possible for the given working set unless physical frames are moved to secondary flash memory. Not only such requirement is prohibitive for energy-efficient embedded system, but also such a cache organization requires tag address translations each time the cache is accessed resulting in significant power consumption.

The technique offers a cache architecture where both virtual and physical tags are utilized at the same time. Virtual tags are used for the majority of memory references to non-shared data, while physical tags are used only for references to shared memory. No restriction in terms of synonym alignment is imposed to the OS memory manager. A special mode bit is associated to each cache line to indicate whether a virtual or a physical tag is being used for that cache line. All the cache lines are virtually indexed, with non-shared data tagged with virtual tags while

shared data references are tagged with physical tags. Figure 3.22 illustrates an example where two processes share a data block, while each one of them having its private data. The shared blocks are identified in advance, in a way described in a subsequent section, and physically tagged when placed into the cache from physical memory. The private data blocks are tagged with extended virtual tags. The physical tag can be translated in parallel with the cache indexing. Thus significant amount of power for address translation for non-shared data references is saved, with practically no performance degradation.

As a virtually tagged cache line can overlap with a physically tagged cache line, the mode bit is used to differentiate between them even if it happens that the virtual tag is identical to the physical tag - for all practical purposes the mode bit can be thought of as an extension to the tag. Consequently, when a sequence of references to private data is being generated by the processor and all of the data is in the cache, no address translations will be needed to access the cache. In the case of a cache miss, the address for the missed reference need to be translated in order to access the physical memory. Furthermore if the replaced cache line is virtually tagged but dirty (in the case of a write-back cache), it needs to be translated into a physical address before writeback to memory.

3.2.3 Write-Back and Write-Through Caches

Write-back and write-through organizations need to be treated differently by the proposed methodology as they treat differently the write accesses to memory.

For write-back caches, the proposed methodology works as explained so far. Cache lines containing data from shared memory regions utilize physical tags. For the rest of the memory, virtual tags are used for both read and write lookups to the data cache.

When a cache miss occurs, one or two address translations are needed. In the case of read-only cache lines being replaced, only one translation is needed for the reference which is being brought to the cache. However, in the case of replacing a modified cache line, two address translations are needed. The first translation is to obtain the physical address for the read data and the second for the physical address of the modified cache line that needs to be stored back to main memory or to the lower level of the memory hierarchy. The translation for the missed reference, of course, has a higher priority and needs to be performed first as the processor is waiting for that data in a way identical to traditional virtually-tagged caches. The second address translation needs to be performed for the just replaced cache line. As writing back to memory is typically not a time critical operation for the processor, it is usually done on the backend of the processor by using a write-buffer implemented as a queue; for instance, the XScale processor features an 8-entry write buffer. The entries in the write buffer are written back to memory when the memory bus is not occupied servicing processor reads. In the organization we propose, the write operations will enter the write buffer immediately with its virtual address and a single bit that specifies a virtual address (a write-buffer entry contains the address and the data to be written). The needed address translation will typically be performed during the next clock cycle; the only exception being that the TLB is

needed to translate another cache miss in the subsequent cycle. In such a case, the write will remain in the write buffer until the TLB is available and its virtual address is translated into a physical one. Having two cache misses in two consecutive clock cycles is an extremely rare situation and even in such a case, the write to memory will be delayed in the write buffer for a cycle, which will not impact the processor performance. The only modification needed to the TLB controller is the simple logic that checks whether there is a write buffer entry carrying a virtual address that needs an address translation. The typical size of a write buffer is four entries and the needed multiplexing for reading these entries already exists as on cache miss the processor first checks the write-buffer entries before going to memory. For the proposed organization checks in the write-buffer are performed in the same way as in the cache by using either a virtual or a physical tag. Consequently, the only hardware overhead to the TLB controller is the logic of utmost several gates that checks the mode bit for the write-buffer entry and subsequently replacing the write-buffer entry address with a physical one.

Write-through caches propagate each write to the lower level of the memory hierarchy. Because of this, in the case of memory write the physical address is always needed regardless of whether the memory location is shared or not. Read memory references are handled in the same way as for the write-back cache organization. Consequently, the proposed technique will not achieve as large energy reductions as in the case of write-back caches. This effect is quantitatively evaluated in our experimental results. Note that this situation is different from the case explained above when two address translations are needed. In this case all the writes, which

hit in the cache, would need to be propagated to physical memory, thus requiring physical addresses (and the corresponding single TLB lookup) regardless of whether they are to private or shared memory regions. In order to alleviate the write effect for write-through caches, we introduce a *Physical Page Latch (PPL)*, which stores the translated address for the most recent memory write. Together with this latch, we also introduce a register, which holds the VPN for that most recently accessed physical page frame. This VPN serves as a tag, which identifies whether the currently write-accessed memory page is the same as the most recently written to page. Such a check is preformed by simply comparing the VPN of the write operation with the VPN stored in the register. In the case of a match, no TLB lookup needs to be performed as the physical page number for that VPN is present in the PPL. In this way, a long series of writes to the same memory page will require only a single TLB lookup for the first write, while all of the subsequent writes will reuse it from the PPL. The only overhead associated with the PPL is the VPN comparator activated on a write to memory. The power needed by such a comparator, which is essentially a collection of XOR gates is extremely smaller, than the power taken by a TLB lookup. In our experimental results we evaluate in details the utility of the PPL to achieve significant power reduction for write-through caches.

3.2.4 Identifying the References to Shared Memory

One of the important aspects of the proposed low-power cache tagging organization is that it needs to be known in advance whether the address generated

by the processor refers to shared or private data for that program. The proposed scheme performs address translations only for shared memory regions. It is important to understand what these shared regions are with respect to the program's address space and when are the addresses of these locations known. In the context of the traditional physically-tagged/virtually-indexed caches and the proposed heterogeneously-tagged caches, a shared memory region is considered to be a physical memory page, which is mapped by the OS memory manager into the virtual address spaces of at least two processes. Such shared physical memory pages exist for the purpose of communicating data between two different processes running in different virtual address spaces; it is controlled and provided by the OS in the from of *Inter-Process Communication (IPC)* facilities, such as shared memory regions or shared buffers for message passing. The shared memory pages, following a request from the application program are mapped to the virtual address space by the OS. Consequently, this inter-process sharing is always established and performed explicitly by the application process and controlled by the OS. The compiler can be easily made aware through special *#pragma* directives as of whether a particular data buffer from the application address space is to be treated as a shared or private.

It is noteworthy to mention that the inter-process memory sharing that requires special attention in terms of caching is completely different from the data sharing that exists in multi-threaded programs. Multi-threaded applications are formed by running multiple lines of control (threads) that execute in the same address space - the address space of the process within which the threads are created. Consequently, the multiple threads belonging to that process share the address space

and any communication between them is to be handled by the programmer with no intervention of the operating system. Such independence from the OS is the major advantage of the light-weight threading as switching between them is usually performed entirely in user space, thus incurring a minimal overhead. From the address space and data caching perspectives, however, all the memory accessed by the multiple threads for sharing or non-sharing purposes amongst them is private for that address space and is completely indistinguishable from the private memory of processes which do not carry multiple threads. No cache synonyms are possible for the thread-level sharing because the threads use the same virtual addresses to access the internally shared data. From the cache point of view, only memory pages, which are mapped across multiple address spaces and can thus be referred to by different virtual addresses can result in cache synonyms.

Since the shared data buffers are explicitly defined by the application process, they can be easily identified by the compiler and the OS. A mechanism is needed, however, to distinguish the virtual addresses referring to these inter-process shared pages with the rest of the virtual pages. One possibility is to use an extra space from the load/store instructions encoding in order to tag the memory instructions which refer to shared pages. For shared data, which is explicitly declared by the programmer and accessed directly, it is a trivial job for the compiler to tag the corresponding load/store instructions. However, if the program uses pointers to access such shared memory pages it becomes significantly more difficult to determine which pointers can access shared pages; in most of the cases a conservative assumptions need to be made and the majority of pointers marked as potential references to shared memory

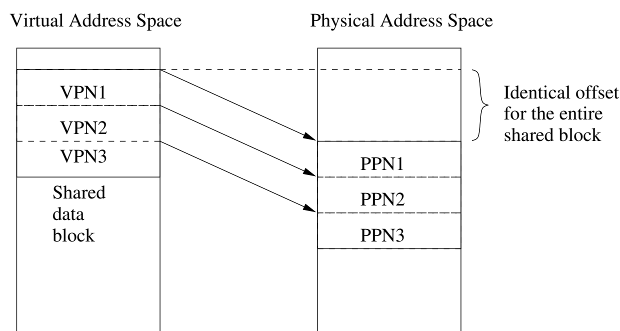


Figure 3.17: Linear mapping from VPNs to PPNs

pages.

An alternative approach, which we have followed and recommend for our technique, is to distinguish the references to shared data through their virtual addresses. The modern embedded processors, such as the Intel XScale and the ARM9, feature a 32-bit virtual address space, which is very large for the demand of any embedded application. In order to distinguish references to shared pages through the virtual address, a portion of that address space may be easily reserved for such pages. For example, a small set of the most significant bits from the virtual address may be used to signify whether a shared page is being accessed. In this way, the inter-process shared buffers will be mapped by the OS into the upper parts of the virtual address space and as such will be trivial to identify at run-time when generated by the processor. For instance, a value of "111" in the three most significant virtual address bits may signal that this is a reference to a shared buffer. The hardware needed for this check is a simple 3-input AND gate, which constitutes a zero overhead for all practical purposes.

3.2.5 Low-Power Synonym Alignment

Preventing memory synonyms to be cached at different cache locations, even in the case of physically tagged caches, requires a special alignment in physical memory referred to as page coloring. The OS memory manager must ensure that for each synonym group the virtual and the physical superset bits are identical. In embedded systems when the physical memory resource is limited, such alignment causes additional constraints to the memory management. In order to eliminate this constraint, we introduce a rapid translation for the superset bits, thus leaving the virtual and physical pages of shared memory regions unaligned. In such cases, the superset bits of each virtual page address are not necessary identical to the superset bits of the physical address. Subsequently, the virtual cache index is no longer identical to the physical cache index, thus it can potentially conflict with other virtual cache indices which do not belong to the same synonym group as was shown in Figure 3.15.

In order to avoid such conflicts, the virtual superset bits need to be translated to the physical superset bits with minimal costs. In embedded applications which are intensive on DSP and numerical computations, the shared data buffers are typically input/output buffers, coefficient tables, or just message passing buffers. For most of the cases, the shared buffer fits within a single page, and in the cases where it spans multiple pages it is common to allocate it in consecutive physical memory addresses and also map it to consecutive virtual address pages. Such consecutive property shows that their PPNs can be computed by adding an offset to their VPN.

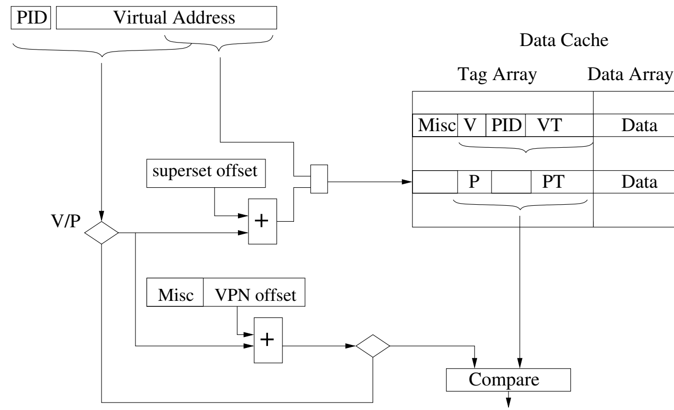


Figure 3.18: Two processes sharing one data block; non-aligned shared virtual addresses

Moreover, the physical superset bits of each physical address can also be converted by adding an offset to the virtual superset bits as show in Figure 3.17. The offset can be determined by the OS when it loads the shared data into the physical memory. Since the width of superset bits is $log_2(cache\ size/(cache\ associativity * page\ size))$, only a few bits need to be manipulated. A fast parallel adder can translate the physical superset bits rapidly with little delay. Figure 3.23 shows an example of two processes as in the previous example but with no alignment property. The physical superset bits are translated with the introduced superset offset adder, and the TLB for page translation is replaced with the page offset adder which is significantly more power efficient. Multiple shared buffers could results in multiple offsets present. For such cases, the multiple offsets can be stored in a very small table (or a small set of registers/latches), the *Synonyms Offset Table (SOT)*, and retrieved before the add operation.

As we pointed out in the previous section, the inter-process shared memory

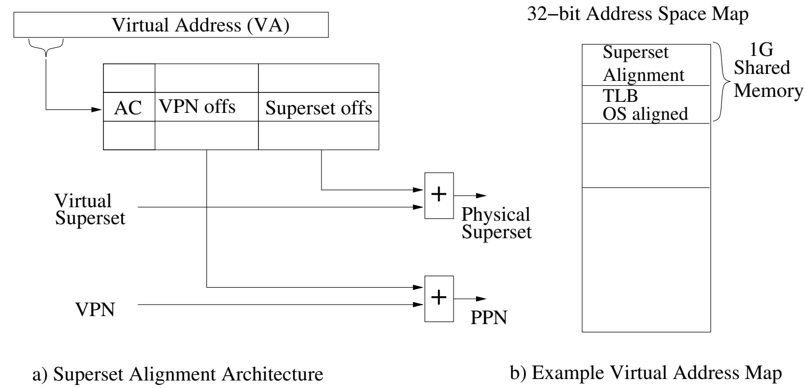


Figure 3.19: The Synonym Offset Table (SOT) and the adders for converting superset bits and VPNs

regions are identified through their virtual addresses. Each such shared buffer is allocated in a pre-specified region of the large virtual address space, which can be uniquely identified by a small subset of the most significant virtual address bits. For example, shared data buffers can be mapped only in the upper half of the virtual address space each such buffer can be placed in a partition, which is uniquely identified by the three bits to the right of the most significant address bit. Consequently, these identifying address bits can be easily used as a direct index into the synonyms offset table to obtain the offsets needed to compute the aligned superset bits as well the physical page number of the shared memory page. The most significant virtual address bit in this example is used as an indication for whether the referenced data belongs to a shared memory page or not.

3.2.6 Hardware Support

The proposed methodology requires a specialized hardware support, which purpose is to perform the different cache access policies for synonym and private references. Changes are needed in the cache line structure and the address translation path to the data cache.

Each data cache line is associated with an additional bit to indicate whether a physical tag for a synonym or a virtual tag for non-synonym reference is being used. To accommodate the virtual tags, the tag field is extended with a Process ID (PID) and Access Controls (AC) bits. The AC bits are transferred from TLB when the cache line is placed from physical memory. Extending the cache tags with PIDs and also associating the AC bits with the cache line is not a requirement freshly introduced by the proposed heterogeneously-tagged cache, but a general requirement of the traditional virtually-tagged caches. Since the proposed technique effectively makes the cache function as a virtually-tagged cache for many references to the cache with no TLB lookups, the small PID and AC extensions to the cache line are required. In our experimental results we take into account the power overhead of these bits.

A very small table (or a small set of registers), the *Synonyms Offset Table (SOT)*, is introduced to capture the few offset constants needed for the superset bit alignment and for address translation for the shared pages. As shown in Figure 3.26a, each SOT entry includes a VPN offset field, a superset offset field, and the access control bits. Each SOT entry represents one shared data block. The SOT

is directly indexed by a few of the most significant virtual address bits. For our experimental results we have assumed 8 entries/registers. As the number of SOT entries bounds the number of shared regions that can be aligned and translated through adders, it is important that there is enough SOT entries for this. However, because of the nature of this inter-process sharing, which typically corresponds to input/output data buffers and buffers for message passing, the number of shared regions is very small and almost always within the range of 2-4. Even though our benchmark did not require so many SOT entries, in order to be conservative in the evaluation of our approach we have accounted for the power overhead of an 8-entry SOT.

The number of SOT entries establishes an upper bound of the number of shared regions per process that can be handled with the adder-based synonym alignment. If the number of such regions exceed the number of SOT entries, the remaining shared regions can be handled in the traditional way through the TLB and memory alignment. An example virtual address space layout is shown in Figure 3.26b. The upper quarter of the address space is reserved for shared buffers, which are always mapped by the OS in this part of the address space. Depending on the number of SOT entries, a corresponding number of shared buffers are mapped into the upper half of the shared memory portions of the address space. The remaining shared buffers are mapped in the lower half of that space, thus translated through the TLB and aligned by the OS in physical memory. With such a layout it becomes trivial to distinguish the address to the inter-process shared memory through a single 2-input AND gate connected to the 2 most significant bits of the virtual

address. The 3rd most significant bits is used to determine whether the reference will be aligned through the proposed superset alignment architecture or will use the TLB. Even though the delay of the region identification logic and accessing the very small SOT is trivial, it can be completely hidden as well. Practically all high-end embedded processors support indexed addressing modes where the value of an immediate field or index register is added to a base register in order to compute the effective virtual address. The value of the base register typically contains an address of a memory segment where the accessed data are allocated. The most significant bits, which define the segment's positions remain the same in this process as the index register or the immediate field typically contain only an offset within that segment. In this way the most significant bits of the virtual address, which determine whether the reference is to a shared region are available earlier in the pipeline before the actual effective address computation and as such can be used to index the SOT and obtain the superset offset prior to entering the memory pipeline stage. Consequently, determining the type of memory address and accessing the SOT table do not introduce any performance overhead.

For all non-shared references, the virtual tag is extended with the PID and send to access the data cache block. The access-control bits field is an aggregate of the access control bits for each VPN in the shared memory block. This aggregation relaxes the access right granularity from page level to multiple consecutive pages of a shared memory block. However, it is very rare that different pages in the shared block have different access rights.

The two adders are used to rapidly and energy-efficiently translate the VPN

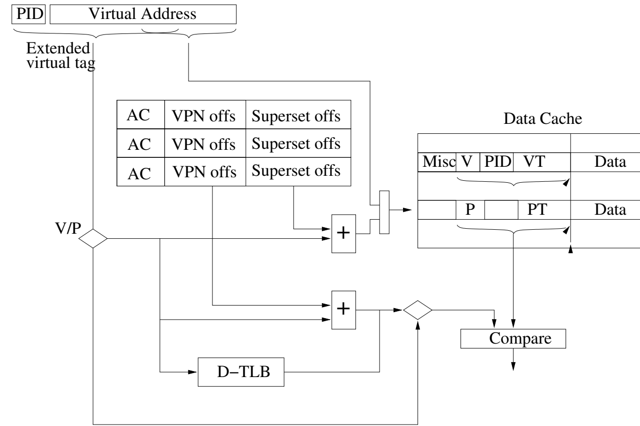


Figure 3.20: Overall hardware organization

and the virtual superset for synonym addresses. The virtual superset bits are added to the superset offset to form an index to the data cache. The VPN tag is added to the VPN offset and the translated physical tag is send to the tag comparator in parallel to the cache indexing. The width of the VPN field is usually much larger than that of superset bits. For 32 bit address and 4k page size with the 16k cache column size, the VPN is 20 bits and the superset is 2 bits. Consequently, the delay of the superset offset adder is much smaller than the delay of the VPN adder. As only the small (typically 2-bit wide) superset adder is on the cache access path, the introduced delay is extremely small. The longer VPN to PPN adder, which replaces the traditional TLB lookup, is on a path parallel to the cache indexing, hence introducing no performance overhead. In any case, the adder delay is significantly smaller than the TLB delay, which it replaces. In our experimental results we have taken into account the extra energy introduced by the superset and the VPN adders. The entire hardware architecture of the proposed approach is presented in

Figure 3.27.

3.2.7 Experimental Results

In evaluating the proposed technique, we have performed a quantitative analysis and comparison between baseline data cache with default TLB structure and the proposed architecture of heterogeneously tagged cache. We have evaluated both the adder-based translation and a traditional TLB-based translation for the shared memory regions. The baseline assumes aligned synonyms, thus the reported advantages would be even larger in reality as handling non-aligned shared memory pages requires extra hardware or software support in the baseline architecture.

In our experimental study we have explored two major configuration. The first configuration is representative for the Intel XScale processors and consists of 32K data cache and 32-entry fully associative TLB; both direct-mapped and 4-way set-associative data cache organizations have been evaluated. The second configuration, which corresponds to the ARM 920T processors, consists of 16K data cache and 64-entry fully associative TLB. Similarly to the first configuration, both direct-mapped and 4-way set-associative data cache organizations have been evaluated. The virtual page size is kept conventionally to 4K size. The energy for each access to data cache and D-TLB are estimated using the Cacti-4 [109] tool, with process technology of $0.18\mu m$.

Table 3.6 shows the dynamic energy (per access) and the leakage power for the baseline cache architectures, the TLBs, and the heterogeneously tagged cache

	32-DM	32-4SA	16-DM	16-4SA	TLB-32	TLB-64
Dynamic (nJ)	0.06845/ 0.07072	0.29496/ 0.29825	0.06051/ 0.06084	0.18089/ 0.18516	0.08393	0.19222
Leakage (mW)	0.30440/ 0.31323	0.30446/ 0.31207	0.15280/ 0.16183	0.15935/ 0.16338	0.00933	0.01786

Table 3.6: Energy for the baseline and the proposed D-Cache/TLB architectures

architecture. The baseline caches are physically tagged and no special tag extensions beyond the standard ones, such as valid and dirty mode bits, are modeled. For this configuration we have used the default Cacti-4 models. The proposed technique relies on a support for virtual tags, thus requiring the tags to be extended with process identifier (PID) and access control (AC) bits. For this we have added 4 extra bits for PID and 2 bits for AC. The only extra bit that our methodology requires, which is in addition to the typical support for virtual tags, is the V/P-bit that specifies the type of the tag; we have included that bit as well to the total of 7 extra mode bits compared to the baseline cache. To model these bits, we have used the standard support for extending the tags in Cacti-4, which is implemented through a special variable, which sole purpose is to define any tag extensions, if needed. Each entry in Table 3.6 depicts a pair of numbers; the first number corresponds to the baseline cache, while the second one for the heterogeneously tagged cache. The last two columns show the energy numbers for the 32- and the 64-entry TLB that we have used for our experiments. As the proposed technique requires no modifications

to the TLB, a single number is shown. The TLB energy numbers are similarly obtained through Cacti-4. In order to model a TLB structure, we have modified the number of address bits to match the VPN length of 20 bits (4K pages within 32-bit address space). Since the proposed technique reduces dynamic power only, the numbers for the leakage power show the impact of the technique on the leakage. As can be seen from the numbers, because of the slight increase in the tag size, the leakage power is slightly increased. This increase, however, is extremely small; for instance, for the 32K caches the increase is around 2%, while for the 16K caches it is around 2.5%.

We have performed our experimental study on a set of widely used multimedia benchmarks from the Mediabench [103] set. The SimpleScalar [105] toolset is used as a simulation test-bed. All the input and output buffers used by these applications are considered shared memory. In a multitasking environment, these input and output data buffers (speech/image/video frames) would be usually communicated from one process to another. For simulation purposes we have captured the address ranges of these shared memory buffers and have provided them to the simulation environment, which was appropriately modified to model the proposed heterogeneous cache tagging. Through architectural simulations, the execution statistics, including the total number of TLB/cache accesses and the number of accessed to the shared memory regions are collected. The overall power consumption is computed by summing the energy needed for all data cache accesses and address translations accounting for the entire application program.

Table 3.12 shows the baseline D-TLB characteristics. The first row in the table

	adpcm	g721	gsm	epic	jpeg	mpeg	mp3
TLB-32	75	4051	4381	640	494	28666	26634
TLB-64	171	9279	10034	1465	1131	65654	60999
32K-DM/32	136	7356	7954	1161	897	52047	48357
32K-4SA/32	338	18290	19778	2887	2230	129410	120235
16K-DM/64	225	12200	13193	1926	1487	86322	80202
16K-4SA/64	332	18011	19476	2843	2196	127437	118401

Table 3.7: Baseline D-TLB and overall D-Cache+TLB energy (in μJ)

contains the benchmark name. The first 6 applications belong to the Mediabench set of benchmarks, while the seventh application is a widely used open source *mp3* encoder. The next two rows report the energy (in μJ) dissipated by the 32-entry and the 64-entry fully associative TLBs only, respectively. The 32-entry TLB is used in combination with a 32KB cache as in the Intel XScale, while the 64-entry is used in combination with a 16KB cache as in the ARM9 architecture. The subsequent two rows show the combined energy of a 32KB cache, direct-mapped and 4-way set-associative, and a 32-entry TLB. The last two rows report the energy of the 16KB cache, direct-mapped and 4-way set-associative, and a 64-entry TLB.

Table 3.8 shows some important execution statistics for each benchmark. All the numbers reported in this table are in thousands. The total number of memory accesses is reported in the first row, while the number of accesses to shared memory

	adpcm	g721	gsm	epic	jpeg	mpeg	mp3
Access	891	48272	52199	7621	5885	341549	317333
Shared	590	295	251	2018	134	93306	4335
Writes	373	11640	11784	841	1620	22924	82971
Private Writes	4	11640	11683	547	1588	21867	80960

Table 3.8: Benchmark cache access statistics (x1000)

regions is shown in the second row. The third row (Writes) show the number of memory writes, while the last row reports the number of memory writes to the shared memory buffers. It can be seen, for example, that for *adpcm* the shared memory accesses are a large part of the total number of accesses, while for *gsm* they are a relatively small part of all the accesses. This can be easily explained by the fact that *adpcm* is not computationally intensive and most of its work processes directly its input and output stream of speech frames. *Gsm* similarly works on a stream of speech frames, however, it is significantly more computationally intensive and for each frame it performs a large number of operations and accesses to the private state of the computation.

We first evaluate the effect on the data cache of using the proposed heterogeneously tagged cache organization for write-back cache policy. The physical address translation uses the default D-TLB. For the memory accesses to non-shared memory regions, the overhead is in extending the cache tag with access control bits and

	adpcm	g721	gsm	epic	jpeg	mpeg	mp3
Enrg(32kD)	113	3441	3713	721	512	34006	24731
Red.(32kD)	33.7/17.1	99.3/53.2	99.5/53.3	71.6/37.9	80.5/42.9	65.6/34.7	91.4/48.9
Enrg(32k4)	315	14422	15590	2450	1851	111203	96545
Red.(32k4)	33.7/6.61	99.4/21.2	99.5/21.2	72.3/15.1	80.6/16.7	67.4/14.1	92.9/19.7
Enrg(16kD)	168	2999	3226	887	582	44319	25680
Red.(16kD)	33.7/25.5	99.3/75.4	99.5/75.6	71.1/54.0	80.2/60.9	64.2/48.7	89.6/68.0
Enrg(16k4)	278	8995	9714	1818	1312	84627	63354
Red.(16k4)	33.7/16.2	99.4/50.1	99.5/50.1	72.2/36.1	80.4/40.3	67.4/33.6	92.5/46.5
Het.-Tag & Adder Tr.:							
Enrg(32kD)	66	3415	3693	551	501	26679	23586
Red.(32kD)	95.7/51.2	100.0/53.6	100.0/53.6	98.2/52.6	82.8/41.1	91.2/48.7	95.7/51.2
Enrg(32k4)	269	14399	15570	2285	1840	104361	95766
Red.(32k4)	95.7/20.3	100.0/21.3	100.0/21.3	98.2/20.9	82.8/17.5	91.3/19.4	95.8/20.4
Enrg(16kD)	57	2939	3177	476	551	25924	21859
Red.(16kD)	98.1/74.5	100.0/75.9	100.0/75.9	99.2/75.3	82.9/62.9	92.2/70.0	95.8/72.7
Enrg(16k4)	168	8939	9666	1423	1283	68322	61258
Red.(16k4)	98.1/49.4	100.0/50.4	100.0/50.4	99.2/50.0	82.9/41.6	92.3/46.4	95.9/48.3

Table 3.9: Energy consumption of proposed organization for write-back caches

process ID bits; no tag translation is performed for such memory accesses. When there is a cache miss or a write-back of a cache line containing a virtual tag, the D-TLB is used to obtain the physical memory address. Note that in the case of replacing a modified cache line associated with a virtual tag, two address translations would be needed - one for the address of the new data which is being brought to the cache and one for the dirty cache line that needs to be stored back to memory. In our evaluation we have taken into account this situations and the corresponding number of address translations have been performed. For potentially synonym memory references, i.e. memory references to shared memory regions, the cache access mechanism is identical to the traditional mechanism with the tag part of the address translated through the D-TLB.

Subsequently, we include the effect of the tag and superset bits translation trough the introduced adder-based physical address computation logic. For private memory accesses, the energy per access is unchanged. However, for accesses to shared memory regions, the translation overhead includes the read from the *Synonyms Offset Table (SOT)* in order to obtain the superset bits offset and the VPN offset. It also includes the two adders to compute the physical superset bits and the PPN. In our experiments we have included the energy consumption of the SOT tables and the two adders, the small (2-3 bit) superset bit translation adder and the wider (20-bit) VPN translation adder. We have modeled the SOT energy through Cacti-4 by specifying a small direct-mapped cache with a data array of 8 entries, each 24-bits wide. This is achieved by modifying the address bits and the bitlines accordingly. The energy for this very small SRAM array have amounted to

	adpcm	g721	gsm	epic	jpeg	mpeg	mp3
Enrg(32kD)	113	4417	4693	764	622	35682	31092
Red.(32kD)	33.3/16.9	75.2/39.9	77.1/41.0	64.8/34.2	58.2/30.6	59.8/31.4	67.5/35.7
Enrg(32k4)	316	15399	16570	2494	1961	112903	102926
Red.(32k4)	33.3/6.5	75.3/15.8	77.1/16.2	65.5/13.6	58.3/12.0	61.5/12.8	68.9/14.4
Enrg(16kD)	168	5236	5472	985	834	48132	40055
Red.(16kD)	33.3/25.2	75.2/57.1	77.1/58.5	64.4/48.8	58.0/44.0	58.3/44.2	66.0/50.1
Enrg(16k4)	279	11232	11959	1918	1563	88519	77931
Red.(16k4)	33.3/16.0	75.3/37.6	77.1/38.6	65.4/32.6	58.1/28.8	61.5/30.5	68.6/34.2
Het.-Tag & Adder Tr.:							
Enrg(32kD)	66	3479	3757	554	508	26789	24000
Red.(32kD)	95.7/51.2	98.4/52.7	98.5/52.8	97.7/52.3	81.4/43.3	90.8/48.5	94.1/50.4
Enrg(32k4)	269	14462	15634	2288	1847	104472	96181
Red.(32k4)	95.7/20.3	98.4/20.9	98.5/21.0	97.8/20.8	81.4/17.2	91.0/19.3	94.2/20.0
Enrg(16kD)	58	3002	3241	479	558	26032	22267
Red.(16kD)	98.1/74.5	99.3/75.4	99.3/75.4	99.0/75.1	82.3/62.5	92.0/69.8	95.1/72.2
Enrg(16k4)	168	9003	9730	1426	1290	68433	61672
Red.(16k4)	98.1/49.4	99.3/50.0	99.4/50.0	99.0/49.9	82.3/41.3	92.1/46.3	95.2/47.9

Table 3.10: Energy consumption of proposed organization for write-through caches

$0.00237nJ$. An alternative implementation for the SOT is to simply use a set of registers, which would consume a similar amount of power. It is evident that the SOT energy is orders of magnitude smaller than the energy of the caches or the TLBs. We have accounted for the power consumption of the both adder circuits by using the data presented in [110]. In that research project the authors have implemented and evaluated in terms of power and performance a number of different parallel adder architectures. They have used a $0.13\mu m$ process technology to implement the adders. For our study we have used the Carry-Select Adder (CSA), which in terms of speed is very close to the fastest adders (SCL and KS) while exhibiting an area complexity close to the one of the traditional ripple-carry adder. The experimental results show that a 32-bit CSA adder has been measured to dissipate $1.78pJ$ per access. Using the Cacti formula for scaling process technologies (linear correlation in gate-size and quadratic in voltage) we have estimated the CSA energy for the 0.18μm technology process of our cache, TLB, and SOT components, and subsequently scaled it down (linearly) to a 20-bit and 3-bit adders. The energy numbers that we have computed thus for our 20-bit and 3-bit adders are $2.37pJ$ and $0.1pJ$, respectively.

Table 3.9 shows energy dissipation for the proposed methodology for writeback cache organization. The table is split into two halves. The upper half reports the total energy for the four cache/TLB organizations enabled with heterogeneous tagging and the energy reductions for these configurations compared to baseline cache + TLB organization. The energy reductions reported are in percentage and for each benchmark we include a pair of numbers. The first number shows the energy reduc-

	adpcm	g721	gsm	epic	jpeg	mpeg	mp3
Enrg(32kD)	82	3527	3756	718	517	34036	25558
Red.(32kD)	75.1/39.9	97.2/52.1	98.5/52.8	72.0/38.2	79.6/42.4	65.5/34.6	88.3/47.1
Enrg(32k4)	284	14509	15633	2448	1855	111257	97392
Red.(32k4)	75.0/15.8	97.2/20.7	98.5/21.0	72.7/15.2	79.7/16.8	67.2/14.0	89.7/19.0
Enrg(16kD)	97	3197	3325	880	592	44363	27380
Red.(16kD)	75.0/56.9	97.2/73.8	98.5/74.8	71.6/54.3	79.3/60.2	64.1/48.6	86.8/65.9
Enrg(16k4)	208	9193	9813	1812	1321	84750	65256
Red.(16k4)	75.0/37.5	97.2/49.0	98.5/49.6	72.6/36.3	79.5/39.8	67.2/33.5	89.3/44.9
Het.-Tag & Adder Tr.:							
Enrg(32kD)	64	3421	3696	551	501	26681	23640
Red.(32kD)	98.4/52.7	99.8/53.	99.9/53.5	98.2/52.6	82.8/44.1	91.2/48.7	95.5/51.1
Enrg(32k4)	267	14404	15573	2284	1840	104364	95821
Red.(32k4)	98.4/20.9	99.8/21.2	99.9/21.3	98.2/20.9	82.8/17.5	91.3/19.4	95.6/20.3
Enrg(16kD)	55	2944	3180	476	552	25925	21907
Red.(16kD)	99.3/75.4	99.9/75.9	100.0/75.9	99.2/75.3	82.9/62.9	92.2/70.0	95.7/2.7
Enrg(16k4)	166	8945	9669	1423	1283	68326	61312
Red.(16k4)	99.3/50.0	99.9/50.3	100.0/50.4	99.2/50.0	82.9/41.6	92.3/46.4	95.8/48.2

Table 3.11: Energy consumption of proposed organization for write-through caches with a latched most-recently written physical page

tion of the address translation logic only. This only include the energy dissipated by the TLB. The second number reports the total energy reduction for the data cache and the TLB. Clearly, the TLB reductions are directly proportional to the number of shared memory accesses and how large of a fraction they are to the total number of accesses. As can be seen from the table, the TLB energy reductions for *adpcm* are around 33% only since this benchmarks features a large number of accesses to shared memory. On the other extreme is *gsm*, which achieves 99% TLB energy reduction as it is heavily dominated by local computations, which do not access the shared input/output buffers. The other benchmarks span the range of 67%-92% TLB energy reductions. The total (cache+TLB) energy reduction depends on both the TLB reduction and also on the energy complexity of the particular cache organization. The reduction for 4-way set-associative cache tend to be smaller than the reduction for direct-mapped caches, as the former cache organization is significantly more power consuming than the latter. It is noteworthy that the proposed heterogeneously tagged organization does not reduce the energy dissipated by the cache - it only eliminates the need for translating the tags for most of the accesses to the cache, thus achieving its power reductions from the significantly reduced TLB energy. We report the total (cache+TLB) energy reductions in order to evaluate what is the energy impact of the proposed technique to the entire cache system.

The lower half of Table 3.9 reports the total energy and the energy reductions for the same cache/TLB configurations but this time using the adder-based translation for the superset bits and tags of the accesses to shared memory regions. The organization of the rows is identical to the upper half of the table. Since the TLB

lookups for the shared memory regions are replaced with the much less energy consuming adder operation, it can be seen from the table that the address translation energy reductions are in the range of 82%-99%. This energy reduction include the energy overhead of the SOT table and the both adder circuits.

Table 3.10 shows the energy dissipation and reductions for write-through caches. For this cache organization there is a cache write-back operation for each memory write operation. Consequently, address translation is required for each memory write. Therefore, for the case of write-through cache organization, the proposed methodology reduces the power on reads to private memory regions only; all the write references and the references to shared memory regions would need to be translated as they either utilize physical tag or need a physical address in order to access the lower level of the memory hierarchy. The organization of Table 3.10 is identical to the previous table. It is similarly split into two halves, where the upper half reports on the basic heterogeneously-tagged architecture, while the lower half shows the impact on the adder-based superset alignment and tag translation. Write-through caches, as expected, result in less energy savings as compared to write-back caches, due to the need to perform address translation for each write operation regardless whether it is to a shared or private memory. The energy reductions for most of the benchmarks are with 10%-25% less than the reductions for write-back caches.

Table 3.11 shows the energy dissipation and reductions for write-through cache with the introduction of the *Physical Page Latch (PPL)* caching of the last address translation as described in Section 3.2.3. For this architecture, the physical address

for the last write memory operation is stored into a special output address latch, accompanied by the VPN, which serves as a tag. Thus if two subsequent writes access the same physical page frame, no address translation needs to be performed for the writes after the first one. The translated physical address is latched and reused for any subsequent writes to the same memory page. The organization of this table is identical to the previous two tables. It is clear from the data in this table that the PPL optimization helps significantly for write-through caches and brings back the energy reductions close to the ones for write-back caches. Interestingly, for the *epic* benchmark write-through caches with the PPL optimization achieve slightly better energy reductions than the write-back cache, which can be attributed to a series of write-backs to the same memory page that can benefit from the PPL and avoid address translations.

3.3 Address Translation through Arithmetic Operations

3.3.1 Introduction

The TLB and page table organization assume no prior knowledge regarding the application virtual access patterns and also no assumption is being made regarding the physical addresses where the pages are placed. Such a general organization is needed for general-purpose processors, but can be significantly refined for embedded processors where the program or the set of programs to be executed is known in advance.

By leveraging the unique embedded system characteristics, we introduce a

methodology for application-driven address translation where most of the TLB lookups are replaced with fast and energy-efficient arithmetic *add* operations. As mapping multiple tasks to the same processor becomes a common case for many embedded systems, light-weight and real-time OS kernels are used. These systems typically allocate and load the physical pages for all the applications during boot time, and subsequently keep the memory map unchanged during the system run-time, and especially within the application hotspots, where most of the execution occurs. In such cases, the address translation scheme can be significantly improved as not only are the application VPNs known but also the physical frames to which these VPNs map are fixed and known after loading the system. Typically, the set of VPNs accessed by the program constitute segments of consecutive VPNs. Such VPN sequences correspond to the various data objects accessed by the application. When the OS allocates the physical pages during system load, it has the freedom to place them at various positions but typically they are allocated consecutively in oder to maximize the utilization of physical memory. If the translation architecture is to exploit this knowledge, the mapping between VPNs and their corresponding physical pages can be easily established by the addition of fixed offset instead of looking it up in a table.

In this section, we target the sequence of consecutive VPNs mapped to consecutive physical pages. With the help of the compiler and the OS such pairs are identified and the *adjustment constant* enabling the arithmetic transformation from VPN to PPN determined and captured by the proposed translation hardware. Subsequently, during program execution this constant is added to the VPN in order to

compute the corresponding PPN. In this way, a significant amount of power is saved as no accesses to the power-hungry TLB are performed. As an additional benefit of avoiding TLB lookups, such a translation always succeeds as there is no possibility of unpredictable conflicts which typically prevent the TLB from finding the translation entry. In this way, the execution time of the important program fragments is not only improved but can also be statically estimated with higher precision as a part of the *Worst Case Execution Time (WCET) analysis* [95, 111] - an extremely important requirement for any real-time system.

As implementation flexibility is one of the major advantages of using embedded processor cores, it is extremely important that the hardware support for the proposed technique is reprogrammable. In this way, the proposed technique is applied across multiple applications and even across multiple parts of the same program. Consequently, the proposed hardware support includes a set of registers which capture the information regarding the VPN segments and the *adjustment constant* to produce the corresponding physical pages. In this way, by changing the information captured in these registers (usually controlled by the OS or the compiler) we are able to efficiently reprogram the hardware support in order to apply the proposed technique to another program or program phase within the same application.

3.3.2 Arithmetic Address Translation

Multitasking operating systems are extensively used in many embedded applications [6, 112]. To share the limited physical memory and provide protection of the

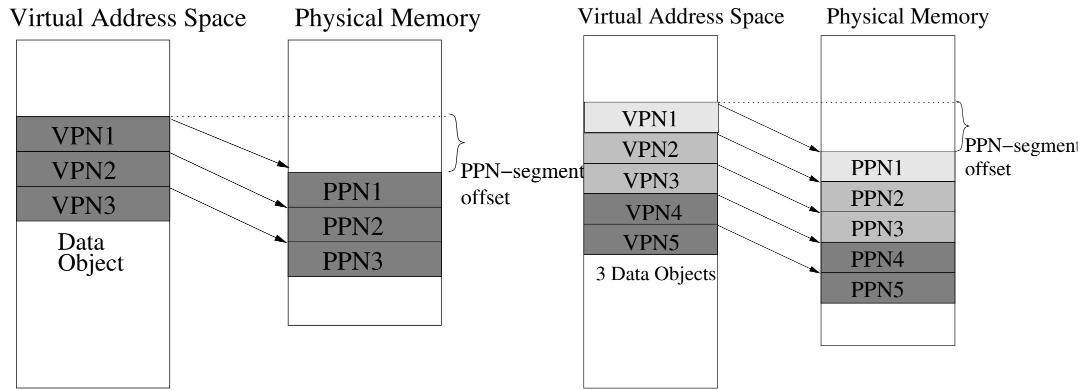


Figure 3.21: A data object forming a VPN-segment

Figure 3.22: Multiple objects forming a VPN-segment

kernel and user tasks, various virtual memory management schemes are utilized. In practice, any real-time OS loads all the tasks to the physical memory at boot/load time, and does not replace physical pages during runtime of the system. The reason for this being that moving physical frames to slower memory might introduce significant delays when these frames are needed later and also introduce unpredictability in the execution times of time-critical parts of the application.

Traditionally, TLB architectures assume no information regarding current physical location (PPNs) of VPNs, i.e. an arbitrary PPN is assumed each time that virtual page is accessed. Consequently, the VPN-to-PPN mapping is captured in a tabular form; the TLB lookup mechanisms uses only the VPN as an input parameter and retrieves the PPN, which can have an arbitrary value. Often times, however, especially in the domain of resource constrained and real-time embedded applications, there is a strong linear correlation between the VPNs and their respective PPNs. For instance, a computationally intensive function or a loop would require accesses to a number of data objects such as data buffers or coefficient ta-

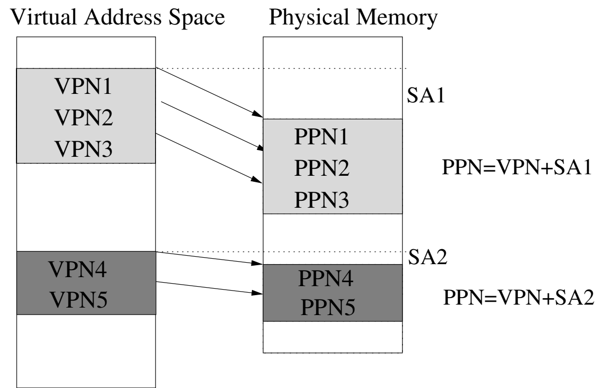


Figure 3.23: Multiple VPN-segments within a task

bles. Each such data object occupies a consecutive address range in the virtual address space of the program, i.e. the object is placed across a set of consecutive VPNs. Even though in theory the corresponding physical frames are allowed to occupy arbitrary locations and migrate during run-time, for practical reasons the set of PPNs is usually fixed during the execution of the critical program function/loop and, furthermore occupies (or can be enforced to occupy) consecutive physical locations. Such sequences of VPNs and their PPNs are refered to as *VPN-segments* and *PPN-segments*. Figures 3.21 and 3.22 illustrate two situations where VPN-segments are mapped to PPN-segments. First, a single data object is shown to form one VPN-segment, while in the second example, three distinct data objects form a single VPN-segment, which is mapped to a single PPN-segment. The latter situation is relevant to closely related data objects, such as coefficient tables and a set of frames from a data stream, which are accessed by an application function or a loop.

Clearly, in situations where data objects reside within VPN-segments, which

in turn are mapped to PPN-segments, the linear correlation between the VPNs and the PPNs can be exploited to efficiently compute the PPNs for each VPN. This computation can be effected by adding a predefined constant offset, which we refer to as a *Segment Adjustment (SA)*. The value of the SA constant depends on the particular physical location at which the PPN-segment is placed. All the VPNs in a VPN-segment can be transformed directly to their PPN by adding the same SA. By avoiding the power consuming TLB lookups and replacing them with fast and energy-efficient addition operations not only is *power significantly reduced* but also *performance is improved* and *made deterministic* since for the majority of load/store instructions a single-cycle address translation can be guaranteed. The net effect on the address translation power is quite significant. For the majority of memory accesses, the address translation would be performed through an energy-efficient adder operation instead of a lookup into a highly associative TLB structure.

When executing a particular part of the program, such as a loop or a function, multiple VPN-segments can be expected to exist. Naturally, each such VPN-segment requires a different SA constant to perform the computation between VPNs and PPNs. Figure 3.23 shows such an example where two VPN-segments exist; the proposed address translation for these two VPN-segments is achieved through the corresponding SA as shown in the figure. The hardware architecture must be prepared to determine to which VPN-segment does the memory reference belong to and then to compute the PPN accordingly.

In order to efficiently implement the outlined address translation technique, a cooperative support from the compiler, the operating system, and the hardware is

required. In general, the set of possible VPN-segments need to be identified during compile-time so that during program load, the OS memory manager is informed about this set of potential VPN-segments and is requested to allocate, if possible, the corresponding PPNs in such a way so that they form PPN-segments as well. A mechanism is also required to associate load/store instruction to VPN-segments, so that the hardware architecture identifies the corresponding SA in order to compute the correct PPNs. The hardware architecture is required to capture the set of SA constants for each active VPN-segment and use them appropriately to compute the PPN. The OS loader is responsible for identifying the correct SAs and to store them into the specialized hardware architecture.

3.3.3 Compiler and OS Support

It has been known that practically any program spends most of its execution times in a few relatively small parts of its code, typically corresponding to loops or functions. Such parts of the code are usually referred to as *phases* of *hotspots* [87, 88]. By targeting the application hotspots, practically all the benefits from the proposed technique can be achieved with minimal OS and hardware cost. Consequently, the proposed technique is applied on the application hotspots, while for the rest of the infrequently executed part of the applications, the general-purpose TLB address translation mechanism is used. Upon entering or exiting a hotspot, the compiler inserts a special setup code which stores certain information into special registers and tables implemented as a part of the specialized hardware support and, thus,

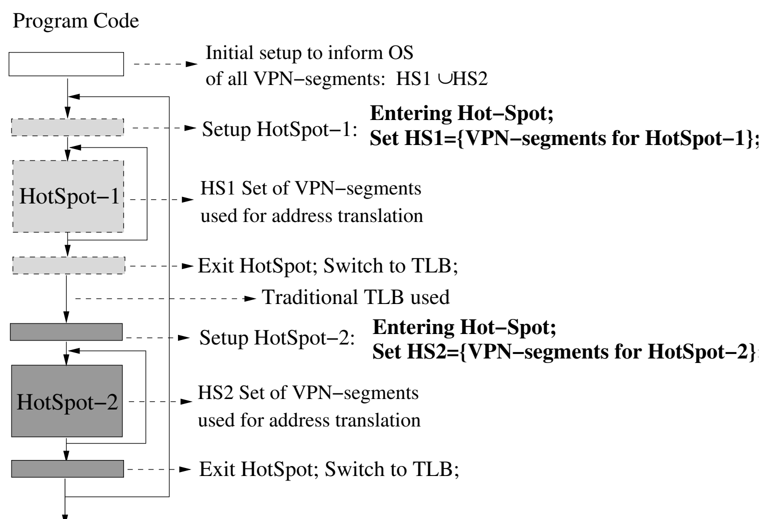


Figure 3.24: Compiler support and setup code insertion

informs the OS and the hardware that a hotspot has just been entered.

The role of the compiler and the profiler in the proposed approach is very important. Initially, the application is profiled in order to identify the hotspots. Each hotspot is targeted by our approach in an independent manner. At that stage after the linking is performed, the set of all VPN-segments is identified. This set consists of groups of consecutive VPNs into which various data objects accessed by the program are mapped. For each hotspots, it is also determined which subset of these VPN-segments is required.

Figure 3.24 illustrates the compiler role in the proposed technique. After the set of all VPN-segments is identified, the compiler inserts a special setup code, which is executed in the very beginning of the program, before the application code is run. The purpose of this setup code is to inform the OS memory manager that the compile/link analysis has identified the provided set of VPN-segments and that the

OS loader must try to allocate the corresponding PPNs in physical memory in such a way so that they form PPN-segments as well. Due to reasons of memory compaction and that the set of distinct VPN-segments is usually small, the OS loader would be able to map most (if not all of them) to PPN-segments. The compiler also inserts a special setup code prior to entering each hotspot. The purpose of this setup code is to inform the OS and the hardware support that an application hotspot is about to be executed; it also informs the OS which VPN-segments are accessed by that hotspot. This information is needed by the OS so that the appropriate SA constants are loaded in the hardware tables for run-time utilization by the arithmetic address translation hardware. The values of the SA constants are initialized at this time with their appropriate values, given the current placement of the PPN-segments. In most cases, these values will not change as the physical frame placement is usually fixed. A similar setup code is inserted on the exit from the hotspot, in order to notify the OS and the hardware that the hotspot has finished and that a default D-TLB address translation must be used from this point.

In order to insert the proper hotspot setup code, the compiler needs to identify the VPN-segments accessed inside the hot-spot. Such a step is requires since due to limited hardware resources only a certain number of VPN-segments would be possible to target. These VPN-segments are selected by giving priority to the ones with the highest frequency of access - an information easily collected from the program profile. When the maximum VPN-segments supported by the hardware is reached, the memory references to the remaining VPNs are assigned to use the default D-TLB. Figure 3.24 illustrates this by showing that the first hotspot is assigned

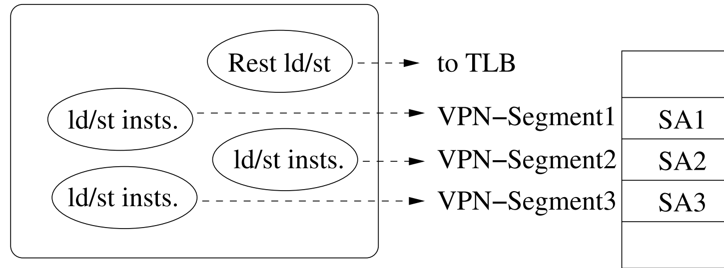


Figure 3.25: Mapping load/store instructions to VPN-segments

the set of VPN-segments denoted as HS1, while the second hot-spot is assigned the set HS2. A priority measure, as already mentioned, could be the total number of accesses to a particular VPN-segment. Another measure could be the size of the segment, since when cached in the default D-TLB, the smaller segment would have fewer VPNs and thus be less susceptible to conflicts than a larger segment with more VPNs. In Section 3.3.6 we see that for most of the benchmarks all of the VPNs inside the application hotspots are a part of a VPN-segment and the total number of VPN-segments per hot-spot is small.

After identifying the VPN-segments within the hot-spots, the compiler needs to provide an identification as of to which VPN-segment does a particular load/store instruction (memory reference) belong to. Figure 3.25 illustrates the structure of this mapping. As the number of VPN-segments is very small per application hotspot, this information can be easily encoded as a part of the load/store instruction encoding. Since the number of segments is limited to only a very few (up to 4 for all practical purposes as our results demonstrate), the number of bits needed to encode the VPN-segments is limited to two or three. Such an identification scheme

implies that each load/store instruction accesses VPNs which belong to the same VPN-segment. This is true for almost any case, and particularly for embedded DSP/multimedia applications, where the application hotspots access a number of arrays or matrices in a regular way. If there is load/store instruction that represent pointer accesses across various VPN-segments, it is assigned to the default D-TLB as this types of memory references cannot take advantage of the proposed approach. Our experimental results, however, demonstrate that such situations are limited especially for embedded applications.

3.3.4 Hardware Support

The proposed methodology requires a specialized hardware support that performs the arithmetic-based address translation and the actions needed for its proper functionality. Special hardware support is needed in order to identify the VPN-segments, to read the SA constant and perform the addition operation to compute the PPN. Consequently, a small table is needed to capture the SA constants for all the supported VPN-segments per application hot-spot. The *Segment Adjustment Table (SAT)* is introduced for this purpose. An entry in the SAT contains an SA constant for a particular VPN-segment. The SAT size is limited to 8 or 4 entries, as for all of the applications on which we have applied to proposed technique, the total number of VPN-segments is well below 8 and in many cases below 4. In our experiments we have evaluated both 4 and 8- entries SAT tables.

Figure 3.26 shows the organization of the SAT table and how its content is

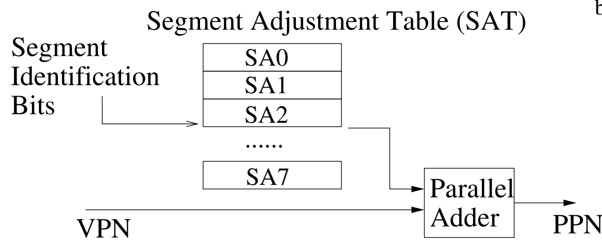


Figure 3.26: SAT table with offsets for PPN computation

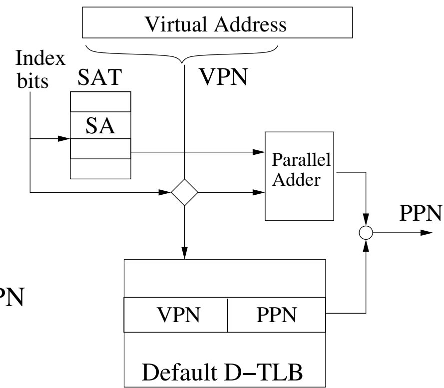


Figure 3.27: Address translation architecture

used to compute the PPN for a given VPN. The special bits identifying the VPN-segment are encoded by the compiler within the load/store instruction code are used to directly index into this small table. As mentioned earlier, in our study we have limited the number of SAT entries to 8/4 as the number of VPN-segments per hot-spots is typically well below these number; our experiments demonstrate that a 4-entry SAT achieves almost all of the reductions that an 8-entry SAT achieves for most of the benchmarks. This is also very beneficial in terms of minimizing the overhead on the load/store instruction encoding. The SA constant read from the SAT table is fed to an adder which adds it to the VPN in order to obtain the physical page frame number. It is noteworthy that the access to the SAT table is performed early in the pipeline right after decoding the load/store instruction. Only the adder is utilized in the pipeline stage where the virtual address is computed and its corresponding physical address determined.

When the PPN-segments are loaded by the OS to the physical memory, their offsets are computed given the starting addresses of the corresponding VPN-segments.

After computing these values, the OS loader places them in a predefined location known to the compiler, so that later on when a hot-spot is to be entered, the setup code that the compiler has inserted loads the SA values into the SAT table.

Figure 3.27 illustrates the entire address translation architecture. The SAT table and the adder are combined with a default D-TLB. The default D-TLB is looked up very rarely and only in the cases where the load/store instruction has been identified to be outside of the VPN-segments supported within the hot-spot. Additionally, when executing outside hot-spots, the default TLB is used for VPN to PPN translation. In Section 3.3.6 we present experimental data covering all these cases while also taking into account the small overhead introduced by the SAT table and the adder circuitry.

It is noteworthy that the lookup into the SAT table is performed early in the pipeline when the load/store instruction is decoded. Therefore, the SAT lookup is outside the critical path of cache lookup or data memory access. Only the adder functionality is needed when the virtual address is generated by the processor to compute the physical address. However, the delay of the adder is orders of magnitudes smaller than the delay exhibited by the D-TLB structure.

3.3.5 System Analysis and Discussion

3.3.5.1 Multitasking Support

Since the proposed approach targets multitasking environments special care needs to be taken to preserve and restore the correct values within the SAT. When

a task switch is initiated by the OS scheduler, it is important that the information regarding the VPN-segments of the currently executing task is preserved. This can be achieved by treating the SAT table as part of the task state, such as the register file and other status registers, which needs to be preserved and later restored when the task is resumed. Since the content of the SAT table is rather small (only 8 or 4 entries), the impact on the context switch time would be negligible. It is also possible that the OS memory manager decides to move some of the PPNs of the suspended task in order to free up space for the physical pages of the new task. In this case, it is important when the old task is resumed that its physical pages are brought back into the main memory in the way so that they form a PPN-segment. This does not impose any extra delay, as these pages would need to be brought back to memory even in the case of traditional address translation architectures. The OS has the degree of freedom to allocate the PPN-segment at different location - in such a case it only needs to compute the new segment adjustment constant and write it into the SAT table.

It is noteworthy that the proposed approach targets a single processor with a virtual memory support, which can potentially execute multiple tasks. Chip multiprocessors and networks-on-chips (NOCs) have emerged as a new implementation platform where multiple processors cores are integrated into a single chip. Nonetheless, in many cases some of these processors would still support virtual memory since often times it is more cost-efficient to allocate a subset of the tasks on a single processor core, especially when there is a significant communication between these tasks and the processor provides sufficient computational power for all of them. The

proposed technique aims at reducing the address translation energy for such cases.

3.3.5.2 Dynamic Memory

Another issue that requires special consideration is dynamic memory allocation, which is rarely used in real-time embedded application. Even if an application requires dynamic memory allocation, such allocation is typically performed outside the hot-spots and only references to these locations are allowed inside the hot-spots. Although virtual addresses for dynamically allocated data are not known at compile time, such data objects are normally assigned by the OS to the heap memory region which occupies a continuous region on the address space. Consequently, heap references can be treated to belong to a separate VPN-segment and a SAT entry can be dedicated for it; the OS determines the SA constant as it allocates the heap. This constant is identical for all the data references to the heap. If the heap VPN-segment grows large and its corresponding PPNs cannot be allocated as a PPN-segment, then the heap references would be assigned to the default D-TLB for address translation.

3.3.5.3 Real-Time Performance

The proposed technique additionally improves on the worst-case timing analysis for real-time applications. All the load/store instructions mapped to VPN-segments are guaranteed a single cycle translation with performance implications identical to a guaranteed TLB hit. This situation is very similar to the application of scratchpad memories (SPM) where memory references to data allocated in the

SPM are guaranteed a single cycles memory access. Our experimental results show that the majority of memory references are guaranteed single-cycle arithmetic-based translation. Such guarantees will greatly improve the upper bounds of the *Worst Case Execution Timing (WCET)* analysis of the code as for most of the memory reference a single cycle address translation is guaranteed. The static WCET analysis [95, 111] is a traditional compile-time step for any real-time application, which obtains guarantees for the worst-case scenario in executing a particular piece of code. Traditionally, when it cannot be inferred by the WCET analysis that a memory reference will always hit in the TLB, the worst case, i.e. a TLB miss, is assumed. For the proposed methodology, however, the majority of load/store instructions (and in many cases all of them) can be guaranteed a single cycle address translation, hence greatly improving on the upper bounds of execution times produced by WCET analysis.

3.3.6 Experimental Results

In evaluating the proposed technique, we have performed a quantitative analysis and comparison between baseline D-TLB structure and the proposed mechanism for arithmetic-based address translation. The baseline D-TLB has 64 entries, with either 4-way or 8-way associativity. The virtual page size is kept conventionally of 4K size. The energy needed to access the D-TLB are estimated using the CACTI tool [113], with process technology of 0.18μ.

In our experimental study we have included the overhead introduced by the

SAT table and the adder circuitry. We have modeled the SAT table as a register file with 8/4 entries. The energy needed to access such a register files is evaluated by using data from [106]. We have accounted for the power consumption of the adder circuitry by using data presented in [114]. We have also adjusted the size of the adder to match the VPN and PPN lengths. For a 32 bit virtual address space with pages of size 4K, the size of the VPN is 20 bits. Additionally, a default TLB with 64 entry is assumed, for VPNs outside the hot-spots or VPNs not covered by our approach due to SAT limitations.

We have performed our experimental study on a set of widely used multimedia benchmarks from the Mediabench set [103] and the MiBench set [104]. The SimpleScalar toolset [105] is used as a simulation environment. After profiling the benchmarks, the hotspots and VPN access patterns are captured through special code inserted to the program prior to simulating the benchmarks to collect run-time statistics regarding the memory references. The final D-TLB power consumption is computed by summing the energy needed for all the VPN to PPN translations.

Table 3.12 shows the baseline D-TLB characteristics. The first row in the table contains the benchmark name. The first 6 applications belong to the Mediabench set of benchmarks, while the seventh application is a widely used open source *mp3* encoder. The last three benchmarks are from MiBench and include one automotive and two security applications. The subsequent two rows show the number of hot-spots identified for each benchmark along with the execution frequency for each hot-spot in percentage. The next two pairs of rows report the energy dissipation in mJ and the number of D-TLB misses for 4-way set associative and 8-way associative

	adp	g721	gsm	epic	jpeg	mpeg	mp3	susan	sha	rij
hotspots	1	1	1	1	2	3	5	1	1	1
freq(%)	58	73	99	75	11,72	81,1 ,11	25,13,24 ,16,18	97	100	66
Energy (4sa)	0.39	21	22.7	3.32	2.57	149	138	0.36	1.23	56.5
Misses (4sa)	8	12	14	2295	2513	1118	46719	26	12	15
Energy (8sa)	0.66	35.7	38.6	5.63	4.35	253	234	0.61	2.08	95.7
Misses (8sa)	8	12	14	2323	2510	1126	53730	26	12	8

Table 3.12: Baseline D-TLB characteristics

TLB organizations, respectively.

Table 3.13 shows the energy dissipation after applying the proposed methodology and using a default D-TLB of size 64 entries. The first row shows the total number of VPN-segments in each hot-spot. As can be seen, most of the hot-spots have fewer than 8 VPN-segments, except in mp3 where some of the hot-spots have more than 8 VPN-segments. We use the VPN access frequency to prioritize which VPN-segments to be covered by the proposed approach; the remaining VPNs are translated through the default D-TLB. The last two pairs of rows correspond to

	adp	g721	gsm	epic	jpeg	mpeg	mp3	susan	sha	rij	avg
VPN Seg.	2	2	3	4	5,7	5,3 ,5	8,11,7 ,7,12	5	4	2	
Energy (4sa)	0.23	9.99	6.58	1.53	0.98	49	41.7	0.11	0.34	29.7	
Red.(%)	42	53	71	54	62	67	70	70	72	47	61
Energy (8sa)	0.34	13.9	6.8	2.1	1.23	56.2	44.8	0.11	0.35	43.1	
Red.(%)	49	61	82	63	72	78	81	81	83	55	71

Table 3.13: Arithmetic-based address translation with an 8-entry SAT

4-way set associative and 8-way set associative default D-TLB. The first row for each pair reports the energy dissipation in mJ after utilizing the proposed technology, while the second rows shows the improvement comparing the reduced energy dissipation against the baseline case. The reported results account for the entire application run including execution outside the hot-spots; the power overhead of the introduced hardware support is accounted for as well. The last column shows the average energy reduction across all the benchmarks; our technique achieves on average 60.8% energy reduction compared to a 4-way set-associative baseline D-TLB and 70.5% compared to a 8-way set-associative D-TLB.

Table 3.14 shows the energy dissipation after applying the proposed methodology with 4 entry SAT. The table rows contain the same information as in the

	adp	g721	gsm	epic	jpeg	mpeg	mp3	susan	sha	rij	avg
VPN Seg.	2	2	3	4	5,7	5,3 ,5	8,11,7 ,7,12	5	4	2	
Energy (4sa)	0.23	9.99	6.58	1.53	1.10	49.3	52.0	0.11	0.34	29.7	
Red.(%)	42	53	71	54	57	67	62	70	72	47	60
Energy (8sa)	0.34	13.4	6.8	2.1	1.47	56.8	65.0	0.11	0.35	43.1	
Red.(%)	49	61	82	63	66	78	72	81	83	55	69

Table 3.14: Arithmetic-based address translation with a 4-entry SAT

table for the 8-entry SAT. The first row shows the total number of VPN-segments for each hot-spot. As can be readily observed this time, since some of the hot-spots have more than 3 VPN-segments, not all the VPN-segments will be covered by the arithmetic translation logic. The set of VPN-segments to be translated through the proposed approach is determined based on their access frequency. The VPN-segments with the highest VPN access frequency are to be covered by the proposed approach; the remaining VPNs are translated through the default D-TLB. Consequently, the default D-TLB will be accessed more often as compared to the case of an 8-entry SAT; thus resulting in smaller energy reduction. Similarly to the previous table, the last column reports the average energy reduction for all the benchmarks. By comparing the average reductions to the previous table, it is evident that the

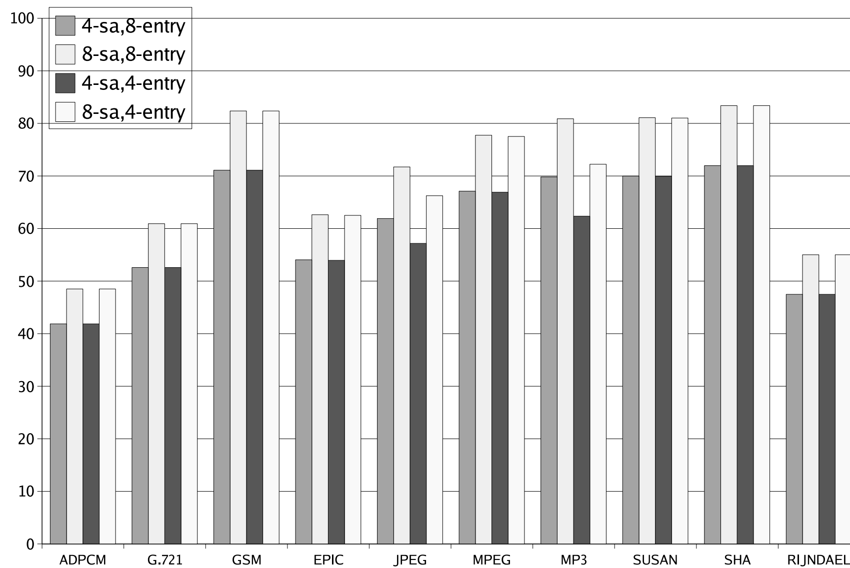


Figure 3.28: Energy reduction comparison

smaller 4-entry SAT table insignificantly lowers the average energy reduction by less than 2%. For most of the benchmarks the reductions are identical and only for *jpeg* and *mp3* a more sizable difference of 9% exists. Consequently, adopting a 4-entry SAT table proves to be practical as the overhead in identifying the VPN-segments is 2 bits only and the achieved energy reductions are extremely close to the reductions achieved by an 8-entry SAT.

Figure 3.28 shows a direct comparison between the energy reductions (in percentage) for the proposed methodology between the 4-way set-associative and an 8-way set-associative baseline configurations. The first two bars in each group represent the energy reductions of an 8-entry SAT case compared to both baselines. The second pair of bars represents the energy reductions of a 4-entry SAT cases

compared to both baselines. It can be noted that the 4-entry SAT case exhibits less energy reductions for the *mp3* benchmark, as this benchmarks has 5 VPN-segments. In the case of a 4-entry SAT one of these segments (the least frequently accessed) is assigned for translation through the default D-TLB.

3.4 Interval Page Table

3.4.1 Introduction

The hardware support for address translation is only one of the system components needed for virtual memory. The system software must maintain a data structure, which purpose is to capture the mapping between virtual pages in the application address space and frames in physical memory. This data structure, referred to as a *page table*, is typically implemented in a table format which occupies a significant amount of memory and is traversed when the hardware MMU cannot provide the physical address due to a miss. The need to perform such page table lookups is the *major obstacle* for providing real-time guarantees.

The traditional page table organizations, as surveyed in [45], have been developed for general-purpose computing systems and servers where it is typical for a very large number of tasks to be simultaneously active. Each task has its own address space and competes for the physical memory with the other tasks, resulting in changes of the memory map. These changes need to be reflected in the page table. Moreover, the tasks compete for entries in the TLB as well. In the case of TLB miss, which as pointed out in [115] can be a very frequent event in many general-purpose

workloads, the page table needs to be traversed to find the requested translation entry.

Two major page table organizations have been established and used in the majority of modern general-purpose processors and operating systems: the *Hierarchical Page Table* and the *Inverted Page Table*. However, the former requires a significant amount of memory, while the latter exhibits highly unpredictable traversal times. Many embedded systems, however, are often memory constrained because of low-cost and low-power requirements and at the same time require strong real-time guarantees. Consequently, none of the traditional page table organizations are well suited for such real-time and memory-constrained systems.

In this section, a novel page table organization targeting real-time and memory-constrained embedded systems is investigated. The proposed organization not only significantly reduces the memory needed to store and manipulate the table but also provides for fast and predictable page table lookups. In many embedded applications, however, the application memory footprint is much less dynamic as dynamic memory allocation and deallocation are extremely expensive operations in terms of performance and power. This property of embedded systems can be exploited, so that the translation information in the page table is stored in a much more compacted form. As page table changes are very infrequent and occur during program load and setup time only, more time-consuming page table insertion/deletion operations can be easily tolerated. The introduced *Interval Page Table (IPT)* represents the mapping between virtual and physical pages in a much more compact way by exploiting the fact that many consecutive virtual addresses are mapped to consec-

utive physical ones and that this mapping changes very infrequently when dealing with real-time and memory-constrained embedded systems. Its compact representation together with application information is exploited to design a very fast and predictable hardware traversal, which completes within a few cycles and requires no system software intervention.

3.4.2 Motivation

In general purpose computer and server systems, various workloads exhibit large variation of memory requirements. Not only the number of active tasks could change rapidly, but also the memory demands of each task could be very dynamic and unpredictable. These properties result in frequent dynamic memory allocations and deallocations. Such activities cause memory pages to be frequently moved to secondary storage and thus later remapped; in addition it also causes the creation and removal of memory mappings from the page table. Consequently, page table lookups and maintenance operations could be very frequent, which results into an optimization goal of speeding up the page table traversal and maintenance operations on average, with page table sizes being not a major concern as large parts of them can be swapped to a secondary storage. Thus fast average lookups and efficient entry manipulation have been the major considerations in page table designs.

In many embedded systems, however, the aforementioned conditions are typically not present. Dynamic memory allocations and deallocations are very expensive in terms of not only performance, but even more importantly *energy* and *real-time*

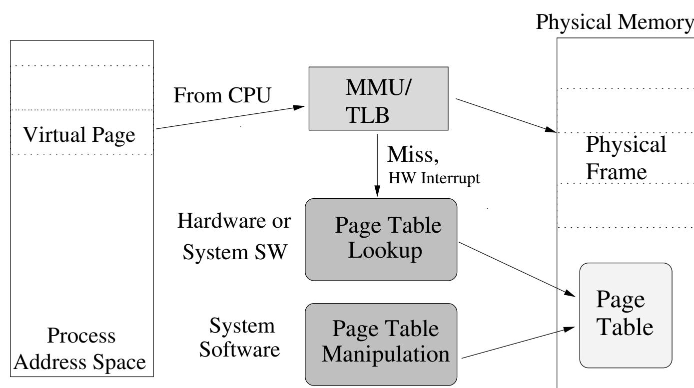


Figure 3.29: Virtual memory architecture

responsiveness and guarantees. The code and data footprints are, thus, quite static and usually no or very limited page re-mapping occurs. Energy-efficiency and real-time guarantees are primary requirements for many embedded systems together with their limited memory resources. Meeting these requirements in the presence of virtual memory has been the major challenge for employing the virtual memory concept in real-time and energy-efficient embedded systems.

Figure 3.29 shows the general organization of virtual memory support. In the case of a MMU miss, a procedure for traversing the page table is executed. This traversal procedure can be implemented either in hardware or executed as a system software routine. The highly-associative MMU translation cache is to be blamed for the excessive power consumption, while the need for a page table traversal in the case of MMU miss contributes to poor real-time behavior. Additionally, due to their large sizes, many of the traditional page table organization are infeasible to adopt in many memory- constrained embedded systems.

Consequently, two major issues need to be addressed when implementing a vir-

tual memory support in an embedded system. First, energy-efficient MMU organizations for address translation are needed. Recent research projects have addressed this problem by introducing new hardware/software address translation schemes, which target embedded applications and their inherent properties of deterministic application knowledge and the resulting possibility for sophisticated compiler involvement, such as [60, 68, 54, 55, 11]. Specialized MMU organizations are introduced which provide deterministic and energy-efficient address translation. Most or all of the needed address translations for a critical program section are pre-loaded and allocated into the MMU in a deterministic way. These techniques significantly reduce the number of page table lookups. However, they cannot guarantee that all memory references will be translated through the MMU. Page table lookups, even though rare, can still occur, and as such need to be taken into account in the real-time worst-case analysis.

The second major problem to be addressed in supporting virtual memory is the page table organization and its contribution to the poor real-time behavior and system cost due to its often excessively large memory space demands. Due to cost and energy constraints many embedded systems lack large secondary storage. In some cases flash memory could be used as such; however, its size is limited and is usually optimized and fine-tuned for the application dataset. In these circumstances, page table size becomes a critical issue in contrast to general-purpose systems. Moreover, the page table traversal needs to performed in a fast and predictable manner in order to meet real-time constraints. None of the traditional page table organizations meet these two fundamental requirements for compact size and real-time traversal

operations.

We introduce the *Interval Page Table (IPT)* organization, a page table structures, which is not only extremely compact in size and significantly smaller than the traditional page tables, but also provides for a very fast and performance-deterministic lookup. The IPT traversal is performed completely in hardware and only the page table maintenance operations, such as inserting or removing an entry, are implemented as part of the system software.

The IPT organization relies on the fact that embedded applications feature a number of important and unique properties as compared to general-purpose systems. The lack of large secondary storage to be used by the virtual memory system to store physical frames currently not used by the application results in page tables, which are mostly static with changes occurring in very rare cases, often during the program load-time only. Therefore, maintenance operations, such as page re-mapping, insertion, and deletion, are very infrequent. The page table organization which we outline in this section follows this principal. Information regarding the virtual and physical memory footprint is used in order to represent the virtual-to-physical mapping in a much more economical way. The set of virtual pages is divided into *page intervals*, where each such interval consists of consecutive virtual pages mapped to consecutive physical memory frames. The proposed *Interval Page Table* consists of an entry for each such interval. The information needed is the interval definition, which includes the starting VPN, the ending VPN, and the access rights for that interval. The only additional data needed to translate a VPN from a given interval is the numerical offset between the virtual interval and its corresponding interval of

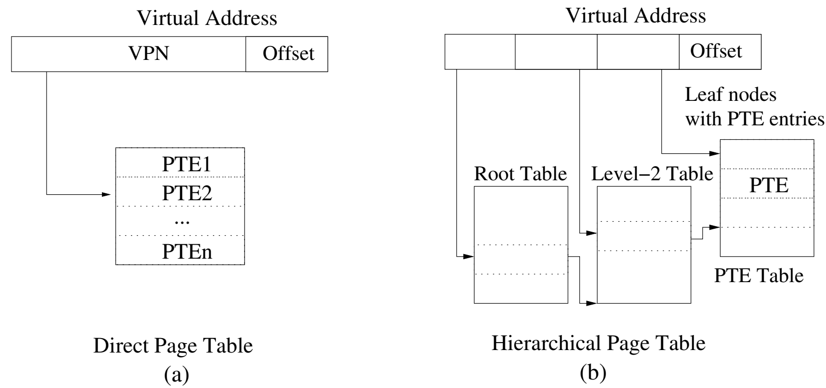


Figure 3.30: Traditional hierarchical page table organizations

physical frames. Consequently, the proposed IPT is highly memory efficient, as it captures information per virtual page intervals rather than virtual or physical pages.

3.4.3 Page Tables Overview

In systems with virtual memory support, the system software is responsible for creating and maintaining the page tables - data structures, which contain the virtual-to-physical mappings. Depending on the page table organization, a table entry, referred to as a *Page Table Entry (PTE)*, may contain a VPN, a PPN, as well as protection and status information.

In systems with relatively small virtual address spaces only a very limited number of virtual to physical mappings need to be captured. Thus a *Direct Table* which contains all the memory mappings is allocated in physical memory, as shown in Figure 3.30a, and directly indexed with the VPN. However, as virtual address spaces increased in size, direct table organizations are no longer capable to capture all the physical memory mappings, as its size quickly exhausts the available physical

memory, even when the application address space contains only a few virtual pages. Several page table organizations have been introduced in order to resolve this problem. These approaches allocate only a fraction of the page table or the page table organization is made to scale with the physical memory, instead of with the virtual address space.

In the classical *Hierarchical Page Tables* the linear virtual address space is viewed as a hierarchical structure of multiple smaller and fixed-size virtual spaces. In this way, the page table is partitioned into a number of hierarchy levels, which are looked up sequentially when traversing the table. The top table, which is known as the *root table*, is placed in a known memory location and contains the address of the second level tables, which in tern capture the addresses of the next levels. The leaf tables, contain the actual PTEs. Figure 3.30b shows the structure of a 3-level hierarchical page table. Only the parts of the hierarchical table, which contain translation information of the current memory map of the application need to be allocated. However, if the program accesses a few but scattered virtual pages, multiple second- and third- level page tables need to be allocated with very low memory utilization. As shown in our experimental results, the hierarchical page tables may require a significant amount of memory.

In the *Inverted Page Tables*, as depicted in Figure 3.31, only the mappings of pages present in physical memory are allocated and captured in the page table. This table organization follows the organization of hash-tables. A hash function is used to compute the index from the VPN. The potential collisions are resolved through the utilization of collision chains within the table. The worst case lookup

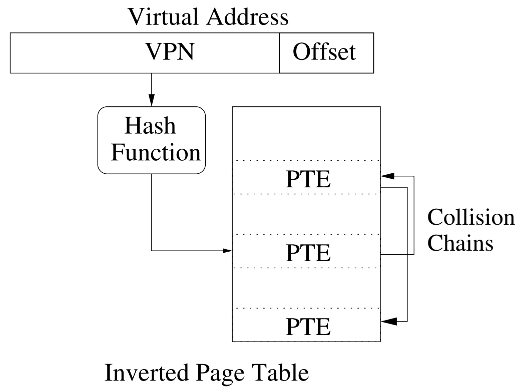


Figure 3.31: Traditional inverted page table organizations

time is determined by the size of the longest collision chain present in the table. The total table size is scaled to the size of physical memory, as the page table contains entries for each allocated physical memory frame. Virtual page number are usually included in each entry in order to resolve the hashing conflicts. In most of the cases, the physical frame number have to be included in the entry as well, in order to overcome problems related to memory sharing and collision minimization. The collision chains are formed either by rehashing or by putting explicit pointers for each entry. Even though the inverted page tables are smaller in size compared to the hierarchical tables, their major drawback in the context of the real-time embedded system is the non-deterministic traversal time. This is due to their organization as hash tables, which can result in rather lengthy collision chains that need to be traversed. The worst-case time analysis needs to take into account the possibility of very long chains, which can result to hundreds of cycles for page table traversal.

As can be readily seen from the review above, none of the traditional page table organizations is well suited for real-time and resource constrained embedded

systems. The hierarchical page tables exhibit deterministic traversal time, however, they usually require an excessively large amount of memory which prevents them from utilization in resource constrained systems. The inverted page tables are relatively less memory demanding then hierarchical tables. However, their hash table structure renders them unusable in real-time systems. The proposed interval page table organization demands significantly less memory than both the hierarchical and the inverted page table, while featuring very fast and time-deterministic traversal times.

3.4.4 The Interval Page Table

The traditional virtual memory implementation assumes no information regarding relations between virtual and physical pages, and no knowledge regarding the physical locations of each page. Consequently, no correlation is assumed between VPNs and their corresponding PPNs. Therefore, the number of page table entries is proportional to the number of pages in either virtual space or physical space. In embedded applications, however, the allocated physical ranges usually conform to certain rules. For example, data arrays are usually allocated to consecutive physical addresses due to considerations of memory compaction and the high cost of page re-mapping. Therefore, if a large data array occupies multiple consecutive virtual pages, the physical pages to which these VPNs are mapped are consecutive as well. Thus there exists a strong correlation between all the mappings of the pages of the data array. Additionally, such sequences of virtual pages usually have identical ac-

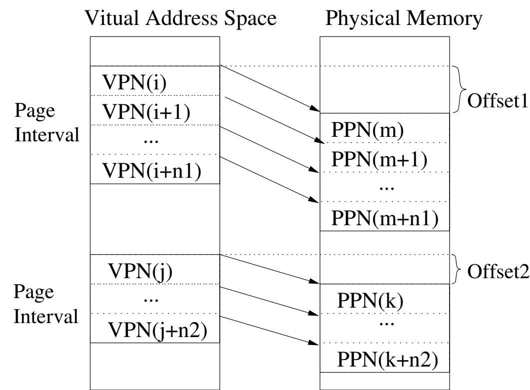


Figure 3.32: Page interval example

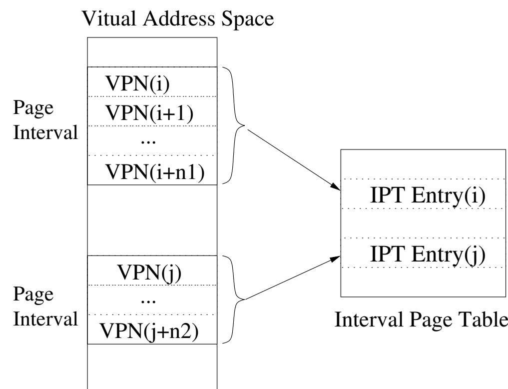


Figure 3.33: IPT: An entry per page interval

cess rights as they correspond to data arrays, and structures that are accessed by the same task, or to an executable binary code.

3.4.4.1 IPT Organization

The proposed interval page table organization considers the virtual address space as a collection of groups of consecutive VPNs which are mapped to their corresponding set of consecutive physical frames. As we explained above and also as observed in our experimental results, such groups of VPNs and PPNs are typical

for embedded systems, and which is even more important change very rarely and are usually defined when the application program is loaded to the system. We refer to such groups of VPNs and PPNs as *page intervals*; examples of page intervals are shown in Figure 3.32. As can be seen from these examples, each mapping from a VPN to its PPN in a page interval has the evident property that the PPN can be computed from the VPN by adding a predefined constant. The value of the constant can be computed by the difference from the first virtual page number and the first physical frame number from the group of pages.

For any page interval, all the physical frame locations can be computed from the virtual page by adding a single constant determined from the corresponding virtual interval. Thus rather than using multiple entries and capture physical locations in each entry, a single combined entry with the interval information and the single offset constant is enough to maintain all the mappings. As shown in Figure 3.33 where multiple such intervals exist, a much smaller page table can be used with each entry corresponding to one interval.

Each entry in the IPT captures the information related to a single virtual page interval, $[VPN_{i1},\ ...,\ VPN_{in}]$. The mapping between the virtual interval and its corresponding physical page interval, $[PPN_{i1}, ..., PPN_{in}]$, is established through the offset constant O_i, which is defined as $PPN_{i1} - VPN_{i1}$ and computed during load time, or by the memory manager when the physical interval or a page is allocated into the physical memory. Consequently, an entry in the IPT will have the following structure: $\{VPN_{i1}, VPN_{in}, O_i, Access - Rights\}$, plus some additional status bits defining the access control rights for this page intervals. All the pages in the interval

are assumed to have identical access control bits, captured within the IPT entry. If different access control bits are needed, the interval is split into multiple intervals each having its own access control bits.

3.4.4.2 IPT Manipulation

The page table is manipulated when entry is added or removed. This mostly happens during load-time when the table structure is created, or in the cases of dynamic memory allocation and deallocation. Because of its complexity, dynamic memory allocation typically occurs outside the critical application regions if it occurs at all. Nonetheless, the page table needs to be able to handle such operations. In order to achieve efficient IPT traversal, the IPT PTE entries are maintained in a sorted order of the page interval VPNs. In the next subsection, we demonstrate how this sorted order is used by the hardware traversal process to rapidly find the required page mapping.

When a new VPN to PPN mapping is to be added to the IPT table, two situations can occur. First, it may be possible to assign the new mapping to an already existing page interval. Clearly, this is the case when the new mapping can be "attached" to the beginning or the end of an existing page interval. In this case, the IPT table entries remain the same and only the information regarding the expanded interval needs to be modified by updating wither the new starting VPN or the new length for that interval. The second situation occurs when the new mapping cannot be assigned to any of the existing intervals. In this case, a new page interval

```
1.   NewMap = new IPT_PTE entry
2.   Interval = FirstIPTEntry;
3.   While(Interval!=LastIPTEntry)
4.     if(NewMap attaches to Interval)
5.       Attach NewMap to Interval; return;
6.     if(VPN(NewMap)<StartVPN(Interval)
7.       CreateNewInterval(NewMap); return;
8.     Interval=NextIPTEntry;
```

Figure 3.34: Adding an IPT mapping

needs to be created and inserted accordingly into the IPT by preserving the sorted order of all the intervals. The pseudo-code for adding a page mapping to the IPT is shown in Figure 3.34. In this pseudo-code, the process of finding the place of the new mapping is performed through a linear iteration through the IPT. This step can be performed to run in logarithmic time, $O(logn)$, with respect to the number of intervals n through binary search. However, the subsequent step of inserting the new interval takes linear time, $O(n)$, to maintain the IPT sorted and thus the total running time would be $O(n)$. For the sake of clarity, in our pseudo-code we have used the linear search procedure for finding the new mapping positions within the IPT. Similarly, when a mapping is to be removed, its page interval is either left intact as in the case when the mapping is positioned at the interval border (either beginning or end of interval), or is split into two new intervals. The pseudo-code for this operation is shown in Figure 3.35. The time complexity of removing a mapping

```
1.  ToRemoveMap = Mapping to be removed;
2.  Interval = FirstIPTEntry;
3.  While(Interval!=LastIPTEntry)
4.    if(ToRemoveMap belongs-to Interval)
5.      if(ToRemoveMap is First-Or-Last-Map in Interval)
6.        RemoveFrom(Interval,ToRemoveMap); return;
7.      else /* ToRemoveMap inside Interval */
8.        RemoveAndSplitInterval(Interval,ToRemoveMap); return;
9.    Interval=NextIPTEntry;
```

Figure 3.35: Removing an IPT mapping

from the IPT is linear, $O(n)$, with respect to the number of page intervals n captured by the table. Similarly to adding a mapping, the linear complexity comes from the situation where an interval needs to be split and the new interval inserted into the sorted table, and possibly requiring the rearrangement (moving with one entry up) of the entire table. Again, for the sake of clarity in the pseudo-code we use a linear search to find the interval to which the mapping that is to be removed belongs to.

It is noteworthy that these two procedures are executed only when there is a need for a change in a virtual-to-physical mapping. This is a situation occurring very rarely, especially for embedded systems, and only when a dynamic allocation or re-allocation has been requested or when the operating system has transferred some of the physical pages to a secondary storage device. Such operations require not only a page table manipulation but also a significant execution time inside the memory

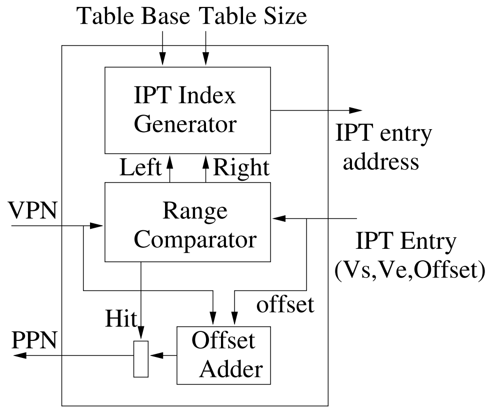


Figure 3.36: Hardware for IPT traversal

management module of the operating system. If existing at all, such operations would only be executed either during the initialization phase of the applications or prior to entering a critical execution section (a function or a loop processing events) that requires real-time guarantees. For the page table lookup operation, the frequency and need of which depends on the capability of the hardware translation buffer, we offer not only fast, but also a time-deterministic procedure.

3.4.4.3 IPT Lookup

Even though page table traversals are rare, especially after the recent offerings for novel MMU organizations for embedded systems, the worst-case real-time analysis must take them into account. Consequently, it is extremely important for real-time embedded systems that this operations are completed in a fast and predictable manner.

The proposed interval page table is traversed by hardware. The IPT page intervals are partitioned into four categories. The *code intervals* capture the instruction

code mappings. The *stack intervals* map the stack pages, the *data intervals* represent the data objects, both statically and dynamically allocated. This partitioning is performed in order to speed up the IPT table lookup. For this reason, the memory reference instructions are tagged with two bits, which identify what type of memory is referred to, i.e. stack, or program data. For the cases, where such ISA tagging is not feasible, the page intervals are partitioned to code and data intervals references to which are easy to distinguish without any ISA modifications.

A binary search procedure is applied within the corresponding IPT partition. The introduced hardware architecture is shown in Figure 3.36. The *IPT Index Generator* provides the access indices for the binary search. It requires two short adders in order to compute in parallel the indices for both the left and the right search paths. Depending on the results provided by the *Range Comparator*, one of these two indices is selected to continue the binary search procedure. The *Range Comparator* consists of two magnitude comparators with width equal to the VPN width. If the IPTs for all the applications are allocated within one very small memory bank, then a one cycle implementation for reading an IPT entry and range comparison will be possible. The two adders within the index generator work in parallel with the range comparator. The *Offset Adder*, which provides the PPN, works in parallel with the range comparator as well. Only when the range comparator finds that the VPN lies withing the current interval, the output of this offset adder is to be provided to the MMU and the processor.

Consequently, the number of cycles needed for IPT lookup depends on the number of page intervals of the corresponding IPT partition (code, stack, data). As

	Total #pgs	Hier #pgs	Hier size	Hier (%)	Inv size	Inv (%)	IPT	IPT (bytes)	Red (Hier)	Red (Inv)
adpcm	4	5	40K	0.0	12K	0.4	3	36	99.9	99.7
g721	6	5	40K	0.1	12K	0.6	3	36	99.9	99.7
gsm	12	5	40K	0.1	12K	1.2	4	48	99.9	99.6
epic	17	5	40K	0.2	12K	1.7	8	96	99.8	99.2
jpeg	20	5	40K	0.2	12K	2.0	5	60	99.9	99.5
mpeg	99	5	40K	1.0	12K	9.7	7	84	99.8	99.3
adpcm /gsm	16	10	80K	0.1	12K	1.6	7	84	99.9	99.3
jpeg /mpeg	119	10	80K	0.6	12K	11.6	12	144	99.8	98.8
adpcm /gsm /g721	22	15	120K	0.1	12K	2.1	10	120	99.9	99.0
jpeg /gsm /gsm	136	15	120K	0.4	12K	13.3	20	240	99.8	98.0

Table 3.15: Interval Page Table (IPT) size and comparisons - 8K pages

	Total #pgs	Hier #pgs	Hier size	Hier (%)	Inv size	Inv (%)	IPT	IPT (bytes)	Red (Hier)	Red (Inv)
adpcm	8	6	24K	0.1	24K	0.4	3	36	99.9	99.9
g721	12	6	24K	0.2	24K	0.6	3	36	99.9	99.9
gsm	24	6	24K	0.4	24K	1.2	4	48	99.8	99.8
epic	34	6	24K	0.6	24K	1.7	8	96	99.6	99.6
jpeg	40	6	24K	0.7	24K	2.0	5	60	99.8	99.8
mpeg	198	6	24K	3.2	24K	9.7	7	84	99.7	99.7
adpcm /gsm	24	12	48K	0.2	24K	1.2	7	84	99.8	99.7
jpeg /mpeg	238	12	48K	1.9	24K	11.6	12	144	99.7	99.4
adpcm /gsm /g721	36	18	72K	0.2	24K	1.8	10	120	99.8	99.5
jpeg /mpeg /epic	272	18	72K	1.5	24K	13.3	20	240	99.7	99.0

Table 3.16: Interval Page Table (IPT) size and comparisons - 4K pages

	Total #pgs	Hier #pgs	Hier size	Hier (%)	Inv size	Inv (%)	IPT	IPT (bytes)	Red (Hier)	Red (Inv)
adpcm	16	7	14K	0.4	48K	0.4	3	36	99.7	99.9
g721	24	7	14K	0.7	48K	0.6	3	36	99.7	99.9
gsm	46	7	14K	1.3	48K	1.1	4	48	99.7	99.9
epic	68	7	14K	1.9	48K	1.7	8	96	99.3	99.8
jpeg	80	7	14K	2.2	48K	2.0	5	60	99.6	99.9
mpeg	396	7	14K	11.0	48K	9.7	7	84	99.4	99.8
adpcm /gsm	46	14	28K	0.6	48K	1.1	7	84	99.7	99.8
jpeg /mpeg	476	14	28K	6.6	48K	11.6	12	144	99.5	99.7
adpcm /gsm /g721	70	21	42K	0.7	48K	1.7	10	120	99.7	99.8
jpeg /mpeg /epic	544	21	42K	5.1	48K	13.3	20	240	99.4	99.5

Table 3.17: Interval Page Table (IPT) size and comparisons - 2K pages

this is known or can be efficiently estimated prior to executing the application program, a careful real-time program analysis can be performed. Because the number of page intervals for each IPT group is typically very small, as evidenced by our experimental results, the number of cycles need for IPT traversal is withing the range of 2 - 3. Not only is such traversal very fast, but the number of cycles for translating the four address groups is known prior to the program execution and available for real-time performance analysis. In the next section we quantify this analysis of the IPT utility with experiments on a number of well known applications.

Because the IPT is looked up very rarely, i.e. less than 0.001% of all the memory references as shown in the next sections, the energy overhead of the introduced traversal hardware is practically non-existent. The area overhead of the introduced hardware is limited to two short adders, 5-6 bit wide, one offset adder which width is equal to the VPN width, usually in the range of 20 bits, and one range comparator, which features two magnitude comparators. We estimate the required die area of this logic to be less than 1% of the total area of a modern embedded processor featuring pipelining, media instructions, and on-chip instruction and data caches.

3.4.5 Experimental Results

To evaluate and analyze the proposed interval page table, we have performed an experimental study with the *M5* full-system simulator [116]. This simulator models a complete computing system with an Alpha-like ISA. The ISA is a classical RISC architecture similar to the ISA used by many modern embedded processor.

The M5 simulator boots the *Linux* kernel v2.6 and executes binaries compiled with a *gcc*-based cross compiler.

We have evaluated the IPT for a set of widely used embedded applications from the Mediabench [103] benchmarks, including speech coders, image, and video processing. We have also created multitasking benchmarks by running simultaneously two or three of this benchmarks. We have modified the Linux kernel in order to monitor the page table operations and to collect statistics regarding the virtual to physical maps and the number of the page table manipulation and traversal operations. The Linux kernel implements a traditional 3-level hierarchical page table; we have included it as a baseline in our experimental results. As another point of comparison we have included a traditional inverted page table organization as well. These two baseline page table configurations are evaluated separately and compared to the proposed IPT. For the hierarchical page table, the total table size includes the sub-tables from all the three levels. We have also provided data regarding the actual utilization of the page table, i.e. the ratio between the size of the actual PTE entries versus the size of the entire table. Similar data is provided for the inverted page table.

To evaluate the proposed interval page table, the page intervals are constructed using information from both the linker map and the Linux memory manager after loading the program. The memory manager provides us with information regarding the actual physical pages used. In this way, the start/end VPNs for each interval together with the offset to its physical pages are identified.

Tables 3.15, 3.16, 3.17 show the experimental results collected on the set of

	Total #Intervals	Code Intervals	Data Intervals	Stack Intervals
adpcm	6	1(1)	1(2)	1(1)
g721	5	1(2)	1(2)	1(2)
gsm	6	1(4)	2(7)	1(1)
epic	44	2(4)	5(12)	1(1)
jpeg	6	2(11)	2(8)	1(1)
mpeg	8	3(10)	3(87)	1(2)
adpcm/gsm	12	2(5)	3(9)	2(2)
jpeg/mpeg	14	5(21)	5(95)	2(3)
adpcm/gsm/g721	17	3(7)	4(11)	3(4)
jpeg/mpeg/epic	58	7(25)	10(107)	3(4)

Table 3.18: Interval Page Table (IPT) entries characteristic - 8K pages

	Total #Intervals	Code Intervals	Data Intervals	Stack Intervals
adpcm	8	1(2)	1(4)	1(2)
g721	7	1(4)	1(4)	1(4)
gsm	7	1(8)	2(14)	1(2)
epic	79	2(8)	5(24)	1(2)
jpeg	7	2(22)	2(16)	1(2)
mpeg	9	3(20)	3(174)	1(4)
adpcm/gsm	15	2(10)	3(18)	2(4)
jpeg/mpeg	16	5(42)	5(190)	2(6)
adpcm/gsm/g721	22	3(14)	4(22)	3(8)
jpeg/mpeg/epic	95	7(50)	10(214)	3(8)

Table 3.19: Interval Page Table (IPT) entries characteristic - 4K pages

	Total #Intervals	Code Intervals	Data Intervals	Stack Intervals
adpcm	9	1(4)	1(8)	1(4)
g721	8	1(8)	1(8)	1(8)
gsm	7	1(16)	2(28)	1(4)
epic	149	2(16)	5(48)	1(4)
jpeg	9	2(44)	2(32)	1(4)
mpeg	9	3(40)	3(348)	1(8)
adpcm/gsm	16	2(20)	3(36)	2(8)
jpeg/mpeg	18	5(84)	5(380)	2(12)
adpcm/gsm/g721	24	3(28)	4(44)	3(16)
jpeg/mpeg/epic	167	7(100)	10(428)	3(16)

Table 3.20: Interval Page Table (IPT) entries characteristic - 2K pages

benchmarks. The three tables report the collected data for memory pages of size 8K, 4K, and 2K, respectively. The first column contains the benchmark name. The first six applications are from the Mediabench set of benchmarks. The next four rows represent groups of benchmarks that are executed in parallel under the control of the Linux kernel. The second column shows the total number of pages mapped and accessed by the application. This set of pages corresponds to the entire memory footprint of the application, including code and data. The next three columns show the characteristics of the hierarchical page table as implemented by the Linux kernel. The first column from the group (*Hier. #pages*) reports the number of memory pages occupied by the page table itself. The next column shows the total memory size of the table, while the last column in the group (*Hier. util.*) reports on the actual utilization of the page table. The utilization is defined as the ratio between the bytes actually used to store page mapping information as compared to the total size of the table structure. The sixth and seventh columns (*Inv. size* and *Inv. util.*) show the total size and the space utilization for an inverted page table. The next two columns, (*#IPT entries* and *IPT size*), report the characteristics of the proposed Interval Page Table. The first one shows the total number of IPT entries required by the application, while the second column reports the total size (in bytes) of the IPT table. Finally, the last two columns report on the space reduction achieved by the IPT as compared to both the Hierarchical and Inverted page table organization. It can be seen, for instance, that the IPT requires consistently less than 1% of the hierarchical page size. It is noteworthy that for many of the applications the hierarchical page table occupies almost as much memory as the memory needed for

	adpcm	g721	gsm	epic	jpeg	mpeg
Total Mem	22791	469293	458618	104036	45734	2308522
64/48 D(I)TLB	$2 * 10^{-5}$	$9 * 10^{-7}$	$1 * 10^{-6}$	$2 * 10^{-5}$	$2 * 10^{-5}$	$2 * 10^{-6}$
48/32 D(I)TLB	$2 * 10^{-5}$	$9 * 10^{-7}$	$1 * 10^{-6}$	$3 * 10^{-5}$	$2 * 10^{-5}$	$2 * 10^{-6}$

Table 3.21: Number of Page Entry Lookups

the entire program data and code.

Tables 3.18, 3.19, and 3.20 show the IPT entries characteristics - it lists all the page intervals classified by their type as well as the number of pages for each interval category. The three tables cover the cases of virtual memory support with page sized of 8K, 4K, and 2K, respectively. The first column lists the benchmarks, while the second column (*Total Intervals*) shows the total number of page intervals identified for each benchmark. The subsequent three columns show the number of intervals and the number of pages for the corresponding interval type (in parenthesis) for code, data, and stack memory pages. It is evident from data in these tables, that the total number of intervals, i.e. IPT entries is quite small even for benchmarks exhibiting large memory footprints, such as the *mpeg*, *jpeg*, and the *epic* applications. From the number of code and data page intervals it can be seen that it takes at most 2 cycles for IPT lookup for code and data, and only data translations for the *epic* benchmark require 3 cycles.

Table 3.21 shows the number of page table lookups for each benchmark. The first row contains the benchmark name. The second row reports the total number of

memory access (x1000). The third and fourth row show the percentage of page table lookup as a fraction of all memory references. The third row shows a hardware TLB combination of 64 entry D-TLB and 48 entry I-TLB, while the fourth row shows a TLB combination of 48 entry D-TLB and 32 entry I-TLB. As shown in the table, the number of page table lookups in these embedded benchmarks is only a tiny fraction of all the memory accesses, yet it still occurs and must be taken into account for real-time guarantees. The fact that this happens extremely rarely results in the zero energy overhead of the proposed hardware IPT traversal scheme.

Chapter 4

Cross-Layer Customization for Multitasking Management

4.1 Introduction

Many modern embedded applications, such as personal organizers, cell phones, and various hand-held devices, constitute complex computing systems where multiple execution tasks cooperate in implementing the product specification. Due to market demands, a large number of capabilities need to be supported, such as aggregated multimedia data processing (speech, audio, video), communication protocols (GSM/CDMA, VoIP, Bluetooth, CAN), security functions, user interfaces, and many others. The utilization of embedded processors for real-time and time-critical control applications have been growing rapidly. The modern automotive industry, for instance, has adopted the approach where tens to hundreds of such processors are used throughout a single automobile [70]. They are used for traction control, anti-lock brake systems, engine control, and many other control and time-critical tasks. Many real-time data acquisition and processing systems such as sensor nodes and networks, impose strict real-time constraints and response time in order to capture, process, and identify rapidly appearing objects and physical phenomena. At the same time, all this processing power needs to be achieved with extremely energy-efficient and low-cost embedded processors.

The inherent multi-tasking nature of these applications has led to implementa-

tions where multiple software tasks are mapped for execution on a high-performance embedded processor such as the Intel XScale [46] and the ARM9 [48], which offer multi-tasking support in the form of *MMUs* and *hardware timers*, and readily available operating systems (OS) which utilize this hardware to implement various forms of multi-tasking.

General-purpose OSs have been known to impose deficiencies in meeting the real-time constraints of many embedded applications. The main reasons for this are the lack of real-time scheduling and the high cost in terms of performance and delay of the context switch procedure. Real-time OS (RTOS) kernels [112, 6] have been introduced in order to achieve more deterministic scheduling of tasks where certain priorities need to be followed. The RTOS scheduler ensures that tasks are scheduled for execution according to their completion deadlines. However, the cost of saving and restoring the task state remains quite high, as this mechanism depends only on the size of the hardware state that needs to be preserved in order to transparently restore the preempted task back to execution. For instance, the process state that needs to be saved on context switch includes program counter, register files, status registers, address space mapping, etc. Therefore, a significant number of memory access operations need to be performed in order to store the state of the preempted task and to load the state of the new task to be executed. To minimize the size of the state, light-weighted multi-threading was introduced [117]. In this approach, a number of threads share some of their state, most commonly their address space. To achieve a context switch, only the register files, and state registers needs to be saved and restored. Due to stringent power constraints the modern embedded pro-

cessors follow the RISC and VLIW paradigms. In these architectures, and even more so in VLIW, the register file is usually large in order to enable the compiler or software developer to exploit instruction level parallelism and maintain high instruction execution throughput; a typical modern VLIW architecture [77] contains general-purpose register files of size from 64 to 256. Even though such register files are clustered, the registers from all the clusters need to be saved on context switch. This implies that it easily takes a few hundred cycles to save and load the general-purpose register file on context switch. Such a significant overhead has two major impacts on the system. First, the system performance is negatively affected, and second, the response time, which is the time between an event triggering a suspended task and the moment when the task resumes execution, is significantly degraded.

In this section, a novel cross-layer customization methodology which significantly reduces the context-switch overhead is investigated. The proposed low-cost context-switch mechanisms are achieved through the active cooperation of compiler, microarchitecture, and operating system (OS). A general-purpose OS or RTOS conservatively saves and restores the entire content of the register file. Such a course of action is needed since no application knowledge is available to the OS task scheduler regarding which registers are alive in that task and thus need to be saved. Similarly, when the task execution is resumed all the register values associated to that task are loaded into the register files. Such a conservative approach has been inherited from general-purpose computing systems, where no prior knowledge regarding the application structure is known, and both the microarchitecture and the OS kernel need to be designed with generality and worse-case assumptions in mind.

We present two complimentary techniques for low-cost and rapid task preemption. Both of them follow the principle that through the close cooperation of compiler, OS, and architecture, very fast and low-cost task switch can be implemented where only a minimal amount of task state is swapped on task preemption. The techniques differ in the way this is achieved; a trade-off is explored between hardware support and the number of cycles needed to perform the context switch. A typical application usually spends most of its execution times in loops or functions, which are generally referred to as *phases* or *hotspots* [87, 88]. Consequently, the code in each such hotspot is typically highly optimized. By applying the proposed methodology independently to each program hotspot all of the benefits from the proposed approach can be achieved with minimal additional hardware.

In the first approach, the *Compiler-driven Context Switch (CCS)*, the compiler identifies what is the minimal number of live registers that needs to be preserved and provides custom software routines to the RTOS kernel. These routines are synthesized by the compiler to save and restore only the live registers for a few *switch* points or basic blocks in the application inner loops and "hot-spot" regions. The switch points/blocks are being optimally identified by the compiler with the property that only a minimal number of general-purpose registers are alive at these points, hence drastically reducing on the overhead of the context-switch procedure. A minimal hardware support is introduced, which is programmed by software and captures the addresses of the switch points and blocks. These switch points/blocks, even though very few, are encountered very often during the program execution as they are fixed by the compiler at positions which are frequently executed and

inside the application inner-loops. The preemption is subsequently deferred to such a switch point/block, where the OS kernel invokes the custom routine for that point/block to store the state and then invokes the custom switch routine for the task which execution is to be resumed.

The second technique introduced in this chapter relies on an introduced pool of extra registers, which are judiciously used by the compiler and controlled by the OS to allocate the minimal set of live registers for the switch points/blocks. We refer to this technique as the *Register Mapped Context Switch (RMCS)*. Similarly, this is achieved through the close co-operation of the compiler, the OS context switch mechanism, and a cost-efficient hardware support in the form of a *limited virtualization of the register file address space*. Compile-time register live analysis followed by a register renaming step actively "packs" the set of live registers into a set of contiguous registers. At context-switch time the OS exploits the limited mapping capabilities of the register file to re-map the set of registers, which are alive during the time of preemption. The effect of the proposed technique is similar to the effect achieved by aggressively replicating the register file to each task and simply switching between the register file replicas during context switch. However, such fast context switch is achieved with a significantly smaller hardware overhead as compared to multiple replicas of the register files. We show that a pool of extra registers consisting of 25% to 50% of the register file is sufficient to provide a context switch with no saving and restoring of general-purpose registers for groups of parallel tasks. When the combined number of live registers for all the parallel tasks exceeds the pool of available physical registers, only then and only for the less critical tasks

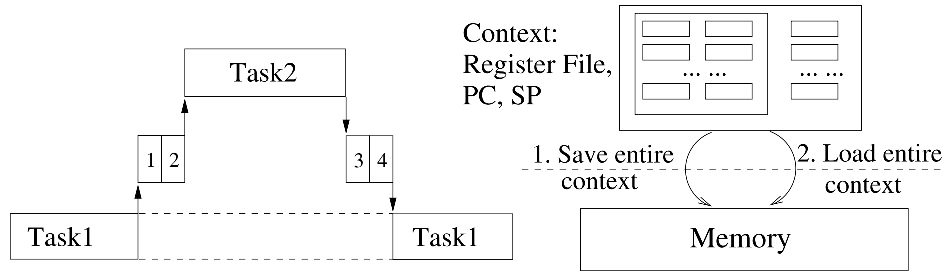


Figure 4.1: Context-switch mechanism for preemptive multitasking.

that exceed the pool of available physical registers, the corresponding parts (pages) of the register files containing live registers will be moved to memory.

4.2 Motivation

In preemptive multi-tasking the OS can pause a low-priority task and assign the CPU to a higher priority task - an OS controlled event referred to as *preemption*. Preemptive multi-tasking relies on a timer to generate interrupts at regular time intervals. When such an interrupt occurs, the execution control is transferred to a kernel routine that determines whether a task switch needs to be performed and, subsequently, to perform the context-switch. This process is depicted in Figure 4.1. When the preemption interrupt occurs, the OS kernel executes two basic procedures in order to perform the preemption. In Step 1 the kernel saves the state of Task1, while in Step 2 it loads the state of the preempting Task2. Switching back to the original task is performed in the same way. When Task2 is preempted, identical pair of steps are executed, denoted as Step 3 and Step 4 in Figure 4.1.

Figure 4.1 illustrates the mechanism of general-purpose task switch in preemp-

tive systems. Before the preempting task can start execution, there are two steps as shown. First, the state of preempted $Task_1$ is saved and then the state of the preempting task $Task_2$ is to be loaded into the processor hardware. When $Task_2$ is brought back to execution, the identical steps but with reversed state are performed. The context switch overhead is the sum of the execution cycles for all these steps of saving and restoring the hardware state. In modern RTOS kernels, deciding which task is next takes only a few cycles in order to lookup a priority queue which does not depend on the number of tasks in the system [118]. In this section, we focus on the cost of the context switch only in terms of execution cycles needed to save and restore the states of the preempted and the preempting tasks.

Reducing the context switch overhead has been the focus of various research projects. The main goal is to reduce the number of load/store instructions needed to save and restore the task context. A simple hardware scheme assigns a separate register file to each task. During task switch, the preempting task immediately starts execution by using its own register file copy. The obvious drawback of this approach is the excessive hardware overhead in terms of an extra register file copy for each parallel task in the system. In practice, a restricted version of this approach is used where the kernel and user-level code operate on separate register files.

Instead of having a distinct physical register file for each task, another hardware solution is to have a relatively small ISA-visible set of registers, while implementing a significantly larger physical register file. This organization is illustrated in Figure 4.2. At run-time, each virtual register is renamed to a free physical register. This approach is very popular in superscalar processors, such as Intel Pentium

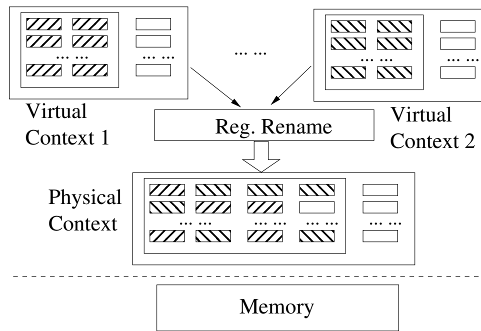


Figure 4.2: Hardware register renaming

4 [79] and Alpha 21264 [80]. Such hardware register renaming is mostly used to exploit the available ILP in the program. Context switches are fast as typically only a small part of the physical register file needs to be preserved in memory. However, due to its per-register granularity and the fact that the renaming hardware needs to be activated at every cycle, the approach suffers from excessive power consumption and as such is not applicable to embedded systems. With a similar objective, in [81] the physical register file is implemented as a cache that captures a large number of virtual registers. Fast access to subroutine and multithread contexts is achieved with a non-trivial power overhead.

The notion of fast context switch point has been first introduced in [82]. Each instruction is marked with a special bit to indicate whether a fast context switch is possible at that point. A fast context switch point is defined as an instruction where all scratch registers are dead. Scratch registers are a subset of all the registers which are caller-saved across function call boundaries; the context switch mechanism saves and restores all the remaining non-scratch registers. Consequently, this is a "all-or-nothing" approach targetting old architectures with rather small register files and

no register windowing. VxWorks [6], on the other hand, provides a special hardware context for interrupt service code in order to avoid preserving the task context, and thus improving responsiveness to various system generated events. In [83], the authors have proposed a Simultaneous Multi-Threading platform with mini-thread execution. This approach, however, introduces a non-trivial hardware overhead. In [84], the authors have proposed to reduce the task context in the static OS by finding the live set of each task and merge the set by using the preemption priority information. The authors in [74] utilize and explore cooperative multi-threading instead of asynchronous preemption. Other research has shown that for some applications with known set of tasks and well known run-time characteristics and interactions, an efficient cooperative multitasking system can be synthesized through software thread integration [85, 86]. In a more dynamic system, however, with preemptive multitasking, the active task may have to be suspended at arbitrary point so that another task is placed for execution. Even though the task switch overhead is reduced, the system responsiveness is limited as the compiler must statically decide on the way tasks are interleaved. In a way, the context switch points are explicitly defined by the compiler or the software developer. The tasks are effectively merged and various optimizations can be performed across tasks.

The task switch customization methodology we propose achieves the determinism and efficiency of co-operative multitasking with the asynchronous and dynamic properties of preemptive multitasking. Application-specific information is utilized at context-switch time to preserve the live portion of the task state either through application-specific software routines or through register file re-mapping. In the

first techniques compiler-generated software routines are used by the context switch mechanism to preserve only the minimal set of live registers during preemption in an application-specific manner. In the second methodology the benefits of hardware and software approaches are combined to achieve the fast context switch of replicated register files by using compiler analysis of application-specific knowledge regarding live registers. The compiler renames the set of live registers into small groups of contiguous registers. The physical register file is extended with a small set of spare registers with limited mapping capabilities. At preemption time the OS maps the small fraction of the register file containing live registers to a subset from the pool of registers assigned to capture the live registers of the preempting task at the moment when it has been previously suspended. The effect of separate register files per tasks is achieved with the hardware cost of slightly increased ($\leq 50\%$) register file.

4.3 State Liveness and Preemption Deferral

The cost of task preemption is largely determined by Steps 1,2,3, and 4, as shown in Figure 4.1. Deciding which task to schedule for execution is the responsibility of the OS scheduler. In modern RTOS kernels, this part can be very fast since it takes only a few cycles to lookup a priority queue structure - an operation which does not depend on the number of tasks in the system [118]. The techniques outlined in this chapter are independent from the particularities of the task scheduler; the focus is on the mechanism of efficiently preserving and restoring the states

of two tasks involved in the preemption operation.

The preempting process starts immediately after Step 2. Initially, the state of the preempted task is saved (Step 1). After that the scheduler spawns or resumes the new task by loading the context of the preempting task and eventually loads the PC with the program counter value for the new task. The switch back operation to $Task_1$ is similar but in reverse order. The context switch overhead is the sum of the instructions from Steps 1 through 4 where the processor resources are used for saving and restoring state instead of executing instructions from the application tasks. The context-switch response interval is the time between the beginning of Step 1 where the RTOS kernel starts saving the state and the end of step 2 where the first instruction of the preempting task is executed. The shorter this switch interval is, the more responsive the preempting task (and the system as a whole) is. This property is of extreme importance to many time-critical control application where a suspended task is resumed due to an event from the environment and the processing of this event needs to start as soon as possible.

4.3.1 Register Liveness Analysis

The timer interrupt which triggers the preemption procedure occurs asynchronously from the application execution, and thus can interrupt the task at arbitrary positions. This implies that the processor state actually utilized by the active task is unknown to the OS kernel. Such a knowledge is impossible to be conveyed to the OS scheduler from the compiler as it needs to be done for each instruction inside

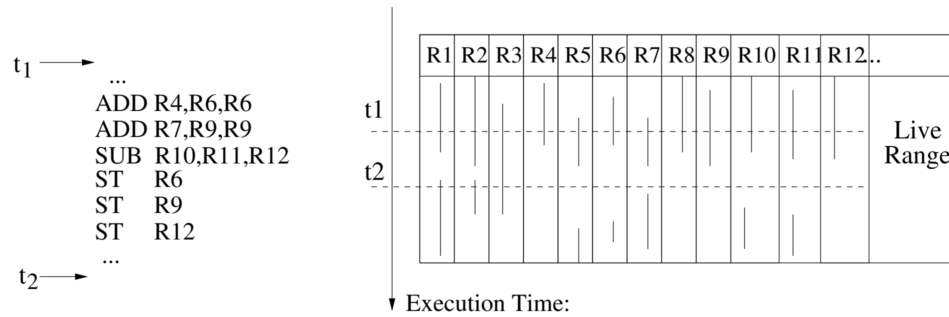


Figure 4.3: Register live ranges and live state

the application - a tremendous overhead which no real system can afford. Therefore, the OS kernel must be conservative in its assumption regarding the actual register utilization and, thus, save all the general-purpose registers. As we mentioned before, in modern processor architectures, such as VLIW, this number could be easily near or above a hundred and result in a significant performance/power overhead during context-switch and deteriorated responsiveness.

If, however, the OS kernel is provided with extra intelligence regarding the actual usage of the processor state, only the live registers need to be saved as part of the task context. During compile-time and especially after the register allocation phase, the compiler has a complete knowledge regarding register utilization. At each instruction position inside the application, the lists of assigned and free (dead) registers are available to the compiler as it is the *register allocation* of the compiler which allocates the physical registers to program variables.

Figure 4.3 depicts an example of a code sequence including the register *live intervals*. The y-axis of the figure corresponds to the instruction sequence (cycles), while the x-axis represents all the general-purpose registers. The vertical lines for

each register correspond to live intervals, which is defined as the time interval between two instructions during which this register is alive. The live interval starts with the register definition by the first instruction and ends with the last usage of that register before it is defined again by a subsequent instruction.

After its last usage and before its subsequent definition by another instruction this register is dead, and thus does not contribute to the task state during this interval. From the example assembly code in Figure 4.3 it can be seen that during the first ADD instruction register R_6 becomes alive (we assume that the destination register of the instruction is the third register). The first ST instruction is the last usage of register R_6 where it is saved to memory. Therefore, after this ST instruction R_6 is no longer alive and it need not be stored during context switch. Similarly, registers R_9 and R_{12} are no longer alive at time t_2. Consequently, for this example at point t_2 three fewer registers need to be saved as part of the task context. Consequently, it can be seen that for this example at time t_1 all registers from R_1 to R_{12} are alive. Therefore if the timer interrupt happens at this moment and context-switch is required, all registers need to be saved. It is evident, however, that if a context-switch is to occur during t_2 instead, only the value of the three live registers, R_1, R_2, and R_3, need to be saved and subsequently restored when the task is resumed for execution later. The savings are quite significant as only 3 out of 32 or 64 registers need to be saved and later restored.

As the number of live and dead registers change at each instruction, it is practically impossible to provide the liveness information to the OS as it would require a tremendous amount of memory. However, if the live information for t_2

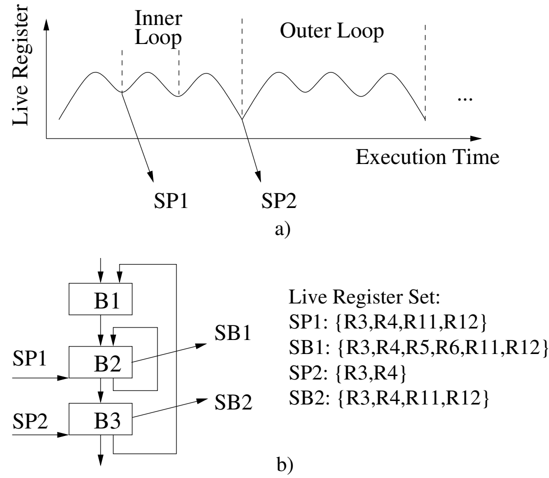


Figure 4.4: Application “hotspot” with two *switch* points and blocks.

only is captured, and the preemption action which happened during t_1 is *postponed* to t_2, a very efficient context-switch can be performed. The time between t_1 and t_2 is spent in executing useful instructions from the preempted application task, which improves on the system throughput. Furthermore, since at time t_2 only a small fraction of the registers is to be saved/restored the response time compared to the general-purpose preemption approach is greatly improved. We present detailed evaluation results in Section 5.6.

4.3.2 Switch-Points and Blocks

A major contribution of the proposed methodologies is the introduction of the concepts of the *switch-point* / *switch-block* and the mechanism of *preemption deferral* until such a point/block is reached in the program. The switch-points are points (instructions) in the program similar to t_2 from the example in Figure 4.3. The switch-points have the property that the set of live registers is minimal. It

is noteworthy that these points correspond to particular program locations, which can be captured by a PC value and no extra instructions are introduced into the application code. The *switch-blocks* is a similar concept but instead of referring to a single instruction, it refers to an entire *basic block* from the application's *Control-Data Flow Graph (CDFG)*. The set of live registers for each basic block, including the switch-blocks in particular, is defined as the union of the live registers for all the instructions within that basic block.

Figure 4.4 shows an example CDFG of an application hotspot consisting of a two-level loop-nest; the innermost loop consists of a single basic block, while the outermost features two basic blocks. As explained above, the number of live registers throughout the CDFG fluctuates and thus exhibit local minimum for some instructions. From Figure 4.4a it can be seen that one such minimum exists in the innermost loop and one in the outermost loop nest. These local minimums points correspond to two switch-points, denoted as SP_1 and SP_2 in the figure. The switch-blocks, denoted as SB_1 and SB_2, correspond to the basic blocks with minimal number of live registers. Typically these are the basic blocks in which the switch-points reside. An example sets of live registers for these points and blocks is shown in Figure 4.4b.

Such a distribution of the switch-points and switch-blocks is representative for a typical loop. This is due to the fact that the scheduled loop, whether it is heavily unrolled or not, typically contains a number of load instructions which load the data to be computed from a number of arrays into a set of registers; it proceeds by the computation, and finishes up by storing the values in these registers back to memory.

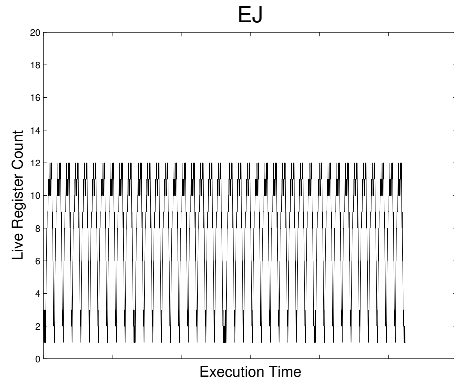


Figure 4.5: Register liveness for EJ

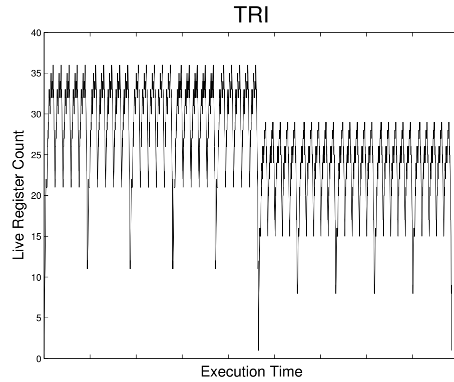


Figure 4.6: Register liveness for TRI

This typical sequence ensures that at the end of the loop only a very few registers are alive and these are the registers carrying a few local variables, including the loop index registers. Consequently, efficient switch-points/blocks are easily identified at the end or at the very beginning of even tightly scheduled loop bodies. The graphs in Figures 4.5 and 4.6 confirm this supposition. These two graphs plot the number of live registers as a function of execution time for the numerical kernels *tri* and *ej* (tri-diagonal matrix transformation, and extrapolated Jacoby transformation - two numerical kernels, which we use as part of our experimental setup). It is evident from this graph that very efficient switch points exist and are easily identifiable as the local minima in the number of live registers. A drop in the number of live registers towards the end of the loop body is quite common and is easily explained with the fact that by the end of the loop all computed values are stored in memory and their registers are no longer alive. These registers will be defined in the beginning of the next loop iteration where the next set of data will be loaded from memory.

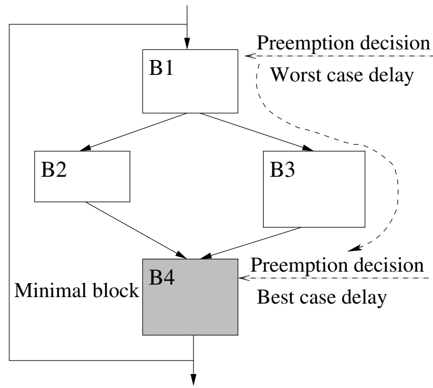


Figure 4.7: Switch-points/blocks placement

4.3.3 Live State Preservation

The process of context-switch is deferred and acted upon only at a switch-point or a switch-block. The decision of whether switch-points or switch-blocks are to be used is based on the trade-off between responsiveness and state amount that needs to be preserved. Clearly, the state amount is minimal at the switch-points and as such less than in the switch-blocks. However, switch-blocks are reachable slightly faster and in some cases can provide better responsiveness even though a slightly larger state needs to be preserved. Since the application hotspots are typically comprised of small amount of code executed iteratively, a very small number of switch points/blocks will be enough to service the context switch mechanism and achieve significant reductions in performance overhead and improve significantly the response interval for each task. As can be seen from the experimental results reported later in the chapter, for the majority of cases only the definition of a single switch point/blocks is enough to achieve these improvements.

Identifying the switch-points/blocks for the application hotspot CDFGs is the

first phase of the proposed technique. These points/blocks are identified statically by the compiler subject that the amount of live registers is minimal. In order not to deteriorate the original response time, the distance between any instruction in the application hotspot to the nearest switch point/block in the dynamic execution flow must not exceed the overhead reduction of the context-switch. In other words, the number of cycles to the nearest switch point/block must be smaller than the number of cycles saved when performing the preemption at the switch point/block. In such a case, even though the preemption might be deferred with several cycles, the response time would be better compared to the general-purpose case. Of course, this is due to the fact that the proposed technique drastically reduces the number of registers that need to be saved and restored during context switch. Figure 4.7 illustrates this important point. Basic block B_4 has been determined to be the single switch-block for the application loop shown by its CDFG. The worst case in terms of preemption deferral corresponds to the situation where the preemption interrupt has occurred during the first instruction of the loop. In this situation, the preemption is differed until the execution reaches the switch-block B_4 at which point a very fast task switch following one of the two techniques described in the following sections is utilized. Clearly, the best case is when the preemption interrupt occurs while the execution is inside the switch-block. In this case the customized task switch is immediately executed for the switch-block B_4. The compiler identifies as many minimal points/blocks as needed in order to ensure that even with a worst-case preemption deferral, the net result is faster than general-purpose preemption. In our experiments we have observed that one minimal point/block per application

hotspot is sufficient for the majority of cases.

Our experiments show that extremely efficient switch points/blocks exist even for applications with high register utilization due to large amount of ILP. Second, due to the limited number of switch points for each application hotspot, the information regarding live registers that need to be utilized by the RTOS kernel is minimal. The two proposed techniques, CCS and RMCS, efficiently solve this problem of transferring the application liveness information in to the OS and preserving/restoring the set of live registers with minimal cost. The CCS techniques uses custom-generates software switch routines for each switch-point/block. These routines are registered with the OS kernel when loading the application and invoked by the kernel. The RMCS techniques uses limited register file mapping to preserve the live state by simply re-mapping the few registers that carry the live state for the switch-point/block.

4.4 Compiler-driven Context Switch (CCS)

The CCS technique relies on the compiler to synthesize a special pair of software routine for each switch-point/block. These routines preserve and restores exactly the minimal set of the live registers at that instruction or basic block. Since the register liveness information is available after the register allocation phase of the compiler, the switch points are determined at compile time. Finding the switch-points and switch-blocks does not introduce any overhead in the compiler, as it can be done simultaneously with the register allocation phase. As the registers are

assigned to each instruction in the generated code, the switch points are dynamically identified as the points in the code with minimal number of live registers. The number of switch-points/blocks depends both on the size of the hotspots and the maximal number of switch-points/blocks which the hardware support allows. The size and structure of the hotspots will determine how many switch points are needed. This is driven by the constraint that a switch-point must be quickly reachable from any point within the hot-spot code as explained in the previous section. As we have observed in our experiments, because of the typical small size of the application hotspots, one or two switch-points/blocks are usually enough for most of the applications to cover all the hotspots. In a subsequent section of the chapter we will describe and analyze the required hardware support for the CCS technique; there we will show that the proposed approach can easily maintain tens of switch-points with minimal hardware and power cost, and no performance overhead.

4.4.1 Compiler and OS Support

Fundamentally, the information regarding the set of live register for each such switch-point needs to be conveyed to the RTOS kernel so that this information is utilized to minimize the task state that needs to be saved and restored. Traditionally, the RTOS kernel features a code which as a part of the context switch module saves the state of the preempted task, including all the general-purpose registers, and load the entire state of the preempting task. One possible approach would be to save a list of live register indices in some memory structure, such as the stack frame of

the task, and have the RTOS kernel use these indices to save and restore only live registers. This approach can be implemented efficiently only if a special hardware, such as a simple DMA-like control unit is used to transfer the set of live registers to/from kernel memory. A software implementation would be quite inefficient as it has to read in the indices and decode them to actual register indices in a "switch"-like statement. The solution we propose through the CCS technique is software only, but instead of storing a set of live register indices and have the RTOS kernel use them, our method has the compiler generate the custom software code to be used to save and restore the set of live registers for each switch point. For instance, if only registers R_1 and R_5 are alive for a particular switch-point, the compiler will generate two special software routines. The first one will simply store R_1 and R_5 to an address inside the stack frame of the task, while the second would load these two registers from the task's stack frame. Prior to entering the hot-spot these two routines will be registered with the RTOS kernel and will be called back as a replacement of the general-purpose RTOS routines for storing and loading the entire register file.

Consequently, for each switch point and its corresponding live register set, one context saving and one context loading software routine is generated at compile time. They are implemented in a similar way as the functions inside the RTOS kernel; there is only store or load instructions in them - no caller save or callee save mechanisms are needed. The target registers are the live register for that particular switch-point/block. The entry addresses of these subroutines are associated with the program counter of the switch point and are registered with the RTOS kernel and the hardware support prior to entering the application hotspot. In order to avoid

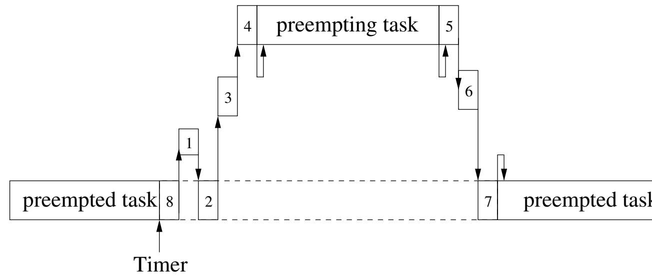


Figure 4.8: Structure of the CCS live state preservation.

any security issue, these routines even though invoked from within the kernel code, where the context switch has been decided, will be executed at the privilege level of the preempted and the preempting tasks. Therefore, these routines can access only memory within the address space of the task and thus expose no security problems. The routines can be executed within the task address space since they use the memory of the task to store the live registers. Clearly, the code of these custom routines is added to the application code. However, these switch routines are very small as they consist of a single load/store instruction per live register. As shown in our experimental results, the number of live registers is extremely small (within 5 - 15 for most of the cases).

Prior to each hot-spot, the compiler would insert a small setup code the purpose of which is to set the processor into a mode of low-cost task switch and register the call-back functions for the switch points within the hot-spot to the operation system kernel. During the preemption phase and the resume phase at runtime, the operation system will use these functions for each switch point to perform the saving and loading of the live registers.

With the introduced concepts of deferred task switch and switch-points/blocks, the preemption interrupt signal from the timer and the actual task switch become decoupled. [1]. The structure and sequences of events of the proposed approach are depicted in Figure 4.8. When the timer interrupt for preemption occurs, it is registered with the interrupt controller but the execution of the preempted task is continued until the switch-point is encountered. Step 8 corresponds to the execution of the preempted application code before reaching a switch-point. As the switch-points are encountered dynamically quite often, this step is rather short; nonetheless it includes the execution of actual application code and is, thus, a useful computation related to the task at hand. While executing the several instructions from the preempted task leading to the switch-point, a special hardware mechanism is activated to identify the occurrence of the switch-point and trigger a signal when this happens. Step 1 is performed at the moment when the switch-point is encountered. The timer interrupt is now acted upon and the RTOS kernel is invoked. This step is identical to the one in the original general-purpose context switch mechanism as was shown in Figure 4.1. Here, the RTOS kernel decides whether a task switch is warranted. If the RTOS scheduler decides that a task switch is needed, the custom software routine generated by the compiler to save the live registers is executed; this corresponds to Step 2. Step 3 is the part of the RTOS scheduler code, which

[1]It is possible, however, in order to minimize the overhead of taking the timer interrupt to program the interrupt controller to defer taking this interrupt only when a switch-point/block is hit. In this way, Step 1 will be executed just before Step 2 and any pipeline flushes associated with taking an interrupt, if present, will be eliminated.

decided which task should be activated for execution. And finally, Step 4, which is the custom code for loading the state of the preempting task is executed. After this routine, the control is transferred to the saved PC of the preempting task. An identical sequence of steps is followed when the new task is preempted in turn from other task or from the original task. The benefits of the proposed approach stem from the fact that Steps 2 and 4 are much shorter and faster than their corresponding steps in the general-purpose task switch mechanism, where the entire register file is saved and restored.

4.4.2 Hardware Support

The purpose of the introduced hardware support is to capture and dynamically identify the switch-points/blocks. This is the only hardware support required for the CCS technique. Since switch-points/blocks are defined within the application hot-spots, inserting instructions to enanle/disable preemption interrupts (in order to completely eliminate the hardware support) would result in non-trivial performance overhead. These instructions would have to be executed at each loop iteration, since preemption interrupts (either timer interrupt or external events) are asynchronous with respect to the program code and cannot be statically predicted at what time during loop execution will occur. The scheme that we propose, instead, does not introduce any new instructions within the application hotspots. Furthermore, the proposed methodology supports any interrupts that result in task preemption, such as timer, special external interrupts triggering specific tasks, etc.

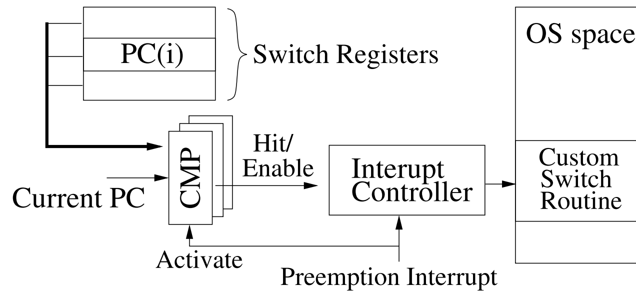


Figure 4.9: CCS Hardware Support

Any such interrupt would triger the detection hardware and be deferred until the next switch-point/block.

Clearly, the switch-points are uniquely identified with the address of the corresponding instruction in the application hot-spot, while the switch-blocks are defined by their start and end address. Here we outline the hardware support for identifying switch-points. The hardware mechanism required for switch-blocks is almost identical - towards the end of this subsection we explain the difference. The addresses of the switch-points defined for the particular application hotspot are stored in a set of special register, which we refer to as *switch registers*. Once a preemption interrupt/request occurs, the current PC must be compared to the each of the switch registers at each clock cycle. When there is a match, a signal triggering a switch-point hit which enables the preemption procedure and initiates the low-cost context switch mechanism at that point.

The structure of the introduced hardware is shown in Figure 4.9. The addresses of the switch points are stored into a set of *switch registers*. This is performed by a code inserted by the compiler prior to entering the application hotspot. These

few instructions are executed only once prior to entering the hotspot. As we have explained earlier and will show in the next section, *one* or at most *two* switch-points are typically enough per application hot-spot for the majority of the applications. Therefore, the area overhead of the introduced hardware support is extremely minimal, as it contains only several switch registers, each 32-bit wide, and a set of comparators for the parallel check with the set of active switch registers. It is noteworthy, that these registers become part of the task state and need to be saved and restored on context switch. We account for these extra cycles in our experimental results. In steady state, when the program executes and no preemption interrupt has been observed, this hardware is disabled. At the moment when a preemption interrupt (timer or other such interrupt) occurs, the circuit is activated and the comparison between the PC and the switch registers is performed from this cycle until a match is found. The preemption is deferred since the interrupt controller is disabled at that moment and the program execution continues. This is the time period which coincides with Step 8 in Figure 4.8. Consequently, the power overhead of this activity is negligently small as this comparison is performed only for several cycles until the switch point is reached. The index of the switch register which matches is provided to the RTOS kernel in order to inform it which switch point has been reached so that the appropriate custom routines for saving the task state are executed.

To support switch-blocks, an additional range comparator is required. A range (interval) check is performed only once when the timer interrupt occurs in order to check whether the PC is currently within the starting and ending address of the

switch-block. If a match is detected, the signal triggering the switch-block hit is set. Otherwise, the task execution continues and the detection of the switch-block becomes completely identical to the detection of a switch-point by using the starting address of the switch-block, which is loaded into a switch register.

4.5 Register Mapped Context Switch (RMCS)

The CCS approach still requires several execution cycles in order to preserve the set of live registers for the switch-point/block. The RMCS technique, outlined in this section, aims at eliminating even these extra cycles by using an additional hardware support. This techniques requires limited mapping capabilities of the register file so that a small part of the register file address space (the first several registers) is virtualized and can be re-mapped at context switch. Compile-time register live analysis followed by a register renaming step actively "packs" the set of live registers into a set of contiguous registers within the mappable parts of the register file. At context-switch time the OS exploits the mapping capabilities of the register file to map the set of registers, which are alive during the time of preemption.

Clearly, the RMCS technique requires more hardware support in the form of extra physical registers and mapping capabilities to a part of the register file. The hardware support required to detect the occurrence of a switch-point or a switch-block, as described in Section4.4.2, is required as well since the actual process of preemption is deferred to such a point or a block. Fundamentally, the only difference between the CCS and the RMCS is in the way they preserve the minimal live set at

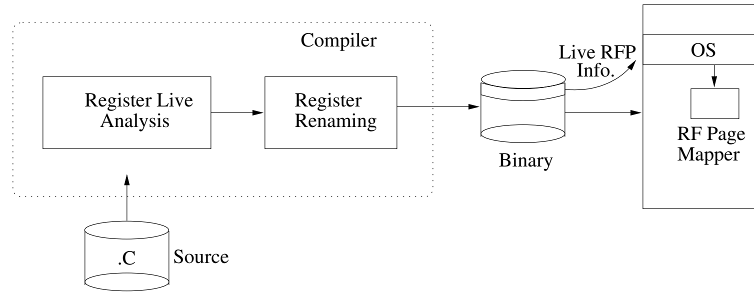


Figure 4.10: RMCS methodology functional overview

the switch-points/blocks. While the CCS technique explicitly saves and restores this set of registers, the RMCS approach keeps the live set in the register file by quickly re-mapping the address space of the register file where the live registers reside. In effect the live register of the preempted task are being hidden, while the live registers of the preempting task are mapped back (mounted) to the register file space which is ISA-visible. As this re-mapping can be achieved in a single clock cycle, the task switch procedure is almost instantaneous. The only small delay incurred is the delay of deferring the preemption until a switch-point/block is reached.

4.5.1 Compiler and OS support

At compile-time the *Control and Data Flow Graph (CDFG)* is being analyzed with respect to register liveness information at basic-block and single instruction levels when the switch-points/blocks are identified. An important compiler phase that needs to be executed for the RMCS technique is that the minimal set of live registers for the switch-points or blocks is subsequently renamed (by the compiler) into consecutive registers residing in the low-addresses of the register file. This

compile-time register renaming phase *has no impact on the performance* as it does not introduce any extra spill/fill code. The register file is extended to contain a pool of extra (spare) set of registers, which can be efficiently and rapidly (in 1 cycle) mapped into the ISA-visible register file address space. In order to further control and minimize the hardware cost for the re-mapping, the register file address space is partitioned into small *Register File Pages (RFP).* By mapping the first several RFPs only into a pool of spare register pages, the context switch procedure is reduced to a single cycle re-mapping event for all the cases where the small set of live registers is accommodated within these pages. Figure 4.10 depicts the design flow and the major steps in applying the RMCS technique.

Figure 4.11 illustrates the organization of the proposed mapped register file. The first several RFPs are mappable through a simple hardware block, which is described in details in the next subsection. For our experiments we have considered RFPs of size 8 and have allowed for only the first 4 RFPs from the register file address space to be mapped. In this way, different physical register pages can be mapped to the ISA-visible register address space. The unmapped pages are not visible to the application code.

The size in registers of the RFPs does not impact the physical organization of the pool of spare registers. As our mapping hardware simply replaces the most significant bits from the register address with an offset within the spare register pool, the size of the RFPs can be easily changed without any modifications to the table of extra registers. This hardware implementation also provides the ability to map any of the spare RFPs to any page from the first four in the register address

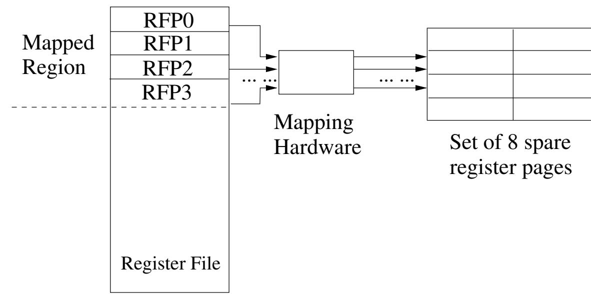


Figure 4.11: Mapped register file organization

space.

At run-time when a preemption event occurs, the special hardware defers the preemption until a switch-point or a block is reached for which the OS is aware about its live register RFPs. Subsequently only these few pages are mapped by executing a single instruction which controls the special register that defines the mapping. As the switch blocks or instructions have a minimal number of live registers, the number of RFPs that need to be mapped is minimal and, as shown in our experimental results, can be achieved for the majority of cases when running several tasks in parallel. It is demonstrated that a small sized pool of extra register pages is enough to simultaneously map the set of live registers for several parallel tasks. The effect of the proposed technique is identical to the effect achieved by dedicating a separate register file to each task and simply switching between these register files during context switch.

During task load-time, the OS can estimate whether the set of live register pages for all the current tasks can be accommodated within the pool of mappable RFPs. If not, depending on the task priority, the non-critical tasks live pages can

be preserved in memory during preemption and not occupy register pages from the pool. For these tasks the OS saves (and later restores) only the RFPs containing the live registers of the tasks. It is clear that for this no application-specific routines are needed as in CCS case - the OS only needs to know which RFPs to save/restore for each hot-spot and switch-point/block for that task. In such a scenario where too many parallel tasks co-exist, the OS would assign the most critical ones to non-swappable register pages in the pool and still take a full advantage of the proposed technique for the time critical tasks.

4.5.2 Hardware Support

Hardware support is required for two components of the proposed methodology. First, an RFP mapping hardware is needed, which would enable the mapping of the first several RFPs in the register address space to be mapped to a pool of extra register pages. The second important hardware block is the module that detects the request for preemption in terms of timer interrupt or external asynchronous interrupt and subsequently defers the preemption point until a minimal block or a minimal point is reached. This hardware module was already described in Section 4.4.2 since it is required for the CCS technique. In this section, we will focus on the mapped register file organization.

Figure 4.12 presents the hardware architecture of the mapped register file. The presented organization assumes 128-entry base register file (ISA-visible) with register file pages (RFP) of size 8; the first four RFPs are mapped to a pool of extra register

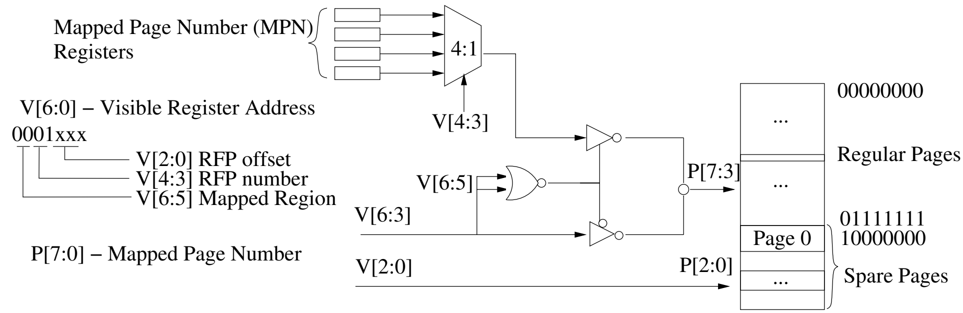


Figure 4.12: Mapped register file architecture

pages. The size of the mapped pages as well as their number can be easily changed - such a change can also be performed at run-time with only a very small modification to the hardware we outline below. The ISA-visible register indices corresponding to the first four register pages can be mapped to either the baseline register file (their normal location) or mapped to the pool of extra register pages, which is a small additional register file. The mapping is implemented by replacing the four most significant bits from the ISA-visible register index with a new 4-bit value (or fewer bits, depending on the size of the pool) that selects a register page from the extra pool. The pool of spare registers can be easily implemented as an additional small register file. As shown in our experimental results, for all practical purposes the size of this additional set of register is within 50% of the basic register file. The 4:1 multiplexer in the figure is responsible for replacing the 4-bit register page number with the 4-bit physical page number. The four 4-bit *Mapped Page Number (MPN)* registers are written by the OS and define the mapping of the current task. Fundamentally, this set of four MPN registers defines the current mapping for the first four register pages from the ISA-visible register space. The values stored in the

four MPNs are analogous to the *physical frame numbers* used in traditional virtual memory to represent the address of the physical memory page. In our experiments we have evaluated both 64-entry and 128-entry register file, which are typical for modern VLIW processors.

The size of the extra register pool determines the number of parallel tasks that can benefit from the proposed technique. If the set of all live registers from the most demanding minimal block in each task can be accommodated within the extra pool, then no saving/restoring of register is needed for context switch for that group of tasks. The only action required by the OS context-switch handler is to replace the values of the four MPN registers. This can be accomplished with a single instructions as the four MPN registers consist of 16 bits total. When the extra pool of registers cannot accommodate all the live pages from the tasks, some of the tasks would be assigned to four pages from the pool, which would be preserved during context-switch. It is noteworthy that at this step the tasks with highest demands for responsiveness can be placed in the extra register pool and utilize the proposed RMCS technique, while the live pages for the less demanding task assigned to the base register file and (only they) preserved at context-switch.

Overhead Analysis. The area overhead of the introduced hardware consists of the pool of extra registers, the mapping logic, and the preemption deferral logic. In our experiments we show that 25% or 50% extra registers is sufficient for achieving zero-cost context switch for several parallel tasks. The mapping logic is fairly small as it constitutes of a 4-to-1 multiplexer, four MPN registers with a total volume of 16 bits and a few gates. The preemption deferral logic consists of a range

comparator and a value comparator per minimal block. The silicon area needed for this is minimal compared to the area of modern pipelined embedded processor with instruction and data caches. In terms of power overhead, the preemption deferral hardware is active only during the few cycles between an interrupt and context switch. Only the 4-to-1 multiplexer is activated during a regular register file access; its power, however, is order of magnitudes smaller than the power needed by a baseline register file only. The minimal delay of the 4-to-1 multiplexer is introduced in the register access path, even though it can be mostly overlapped with the address decoder logic.

4.6 Experimental results

In evaluating the proposed technique, we have performed a quantitative analysis and comparison of baseline general context switch scheme and the proposed application-specific context switch mechanism. The evaluation is performed with the VLIW Example - VEX package [119], which is developed and provided by HP research labs. It includes a state-of-the-art optimizing VLIW compiler and a simulator tool-chain. The VLIW processor core can be configured into various architectures including multiple clusters, register files, and functional units. Each cluster is configured to have two register files, four integer ALUs, two 16*32-bit multiply units, and a data cache port. The cluster can issue up to four operations per instruction. The register set for each cluster consists of 64 general-purpose 32-bit registers. The simulator that comes with VEX is a compiled simulator. For the purpose of simula-

tion, a C code for a custom VEX simulator is generated for the application program. Each VLIW instruction is executed as a call to a function that implements the functionality of the operations within the simulated instruction packet. The number of executed cycles including pipeline stalls and cache misses is maintained captured and reported.

We have evaluated two baseline register files, one consisting of 64 registers and one of 128 registers. We have assumed that the first four pages (each of 8 registers) in the register file are mappable for the purpose of the RMCS technique. To consider the effect of aggressive VLIW compiler optimizations on the proposed methodology, we have included two compiler setups: one where the applications are compiled with heavy loop unrolling and trace scheduling - we refer to this option as an *aggressive optimization*; the second optimization setup includes all scalar optimizations but no loop unrolling - we refer to this as a *scalar-only optimization*.

By instrumenting the compiled simulator for each benchmark, we mark the switch points and model the behavior of the hardware and the RTOS kernel when a switch point is reached. The timer interrupt is modeled by introducing a counter which keeps track of the executed cycles in a way similar to a hardware timer module. When a certain number of cycles is reached, we simulate the occurrence of a context switch. Since in our study we are interested in the responsiveness of the context switch procedure, we measure the delay between the timer interrupt signaling the preemption and the execution of the first instruction from the preempting task. This delay is determined by the preemption deferral interval and the actual cost in terms of cycle for preserving and loading the state of the two participating tasks. These

	ej	lu	tri	mmul	2d-dct	adp	g721	sha	sus
# of h-s	1	2	2	1	2	1	1	5	1
Freq.(%)	100	49,51	54,46	100	49,51	100	100	19,21,20,20,20	100
# SP/SB	1	1,1	1,1	1	1,1	1	5	1,1,1,1,1	1

Table 4.1: Benchmarks characteristics

are the two factors that we take into account in our study.

Prior to instrumenting the compiled simulator, we compile the benchmark application, and the assembly code for the application hot-spots is parsed by a separate script which identifies the register live ranges. The traditional algorithm for this analysis, as described in [120], is used. In a subsequent step, we identify the switch-points and the switch-points as the positions/basic-blocks in the application hotspots where the number of live registers is minimal. We report on the delay and task switch cost reductions for both switch-points and switch-blocks in order to analyze the advantages and disadvantages of both approaches. The baseline architectures constitute a single-cluster and dual-cluster cores. In the single-cluster case, each preemption is assumed to involve 64 store instructions and each resumption has the same amount of load instructions. For the dual-cluster architecture, the registers saved are doubled which results to a total of two groups of 128 instructions. For the CCS approach, we have accounted for the cycles needed by the custom switch routines to execute the context switch. Similarly to the baseline organization, we have assumed a single load/store unit per cluster. Consequently, based on the number of

live registers for each switch-point/block, we have accounted for the corresponding execution cycles introduced by the custom switch routines.

In our experimental study, we have utilized two groups of benchmarks. The computation kernel group includes *Matrix Multiplication (mmul)*, *Extrapolated Jacoby (ej)* method, the *LU matrix decomposition (lu)*, and the *TRI triangular matrix conversion (tri)*; these kernels manipulate large matrices and are extensively used in many algorithms. The application group includes the *2D-DCT* - discrete cosine transform that are widely used in many image and video processing applications, the *adpcm* and the *g721* speech coders from MediaBench[103], the *SHA* hash algorithm, and the *susan* image recognition program from MiBench [104].

Table 4.1 reports the structure of each application benchmark. The first row shows the number of hotspots, while the second row reports the execution frequency of each hotspot. The last row of the table shows the number of switch-points and switch-blocks for each hotspot. In the register live analysis for each application we have found out that the switch-block always contains the switch-point. It is also always the case that the switch-block is always the basic block at the bottom of the hotspot CDFG. As explained earlier, this can be easily explained (and anticipated by us) by the fact that the registers carrying temporary variables as well as data items loaded and stored in memory are dead towards the end of the CDFG and only the registers carrying live variables across the loop iterations are alive.

Tables 4.2, 4.3, 4.4, and 4.5 shows the achieved results. The four baseline architectures correspond to 64 and 128 entry register files, each with aggressive and scalar-only compiler optimizations. The aggressive optimizations include very heavy

	ej	lu	tri	mmul	2d-dct	adp	g721	sha	sus
Live Regs. SP/SB	2/ 8	20/22, 20/27	21/24, 15/17	11/17	14/23, 12/24	11/ 11	{3/5,7/7 8/8,9/9, 12/14}	5/20,11/18, 11/14, 11/18,10/13	20/ 20
CCS State Red.%	97/ 89	72/69, 72/63	71/67, 79/76	85/76	81/68, 83/67	85/ 85	{96/93, 90/90,89/89, 88/88,83/81}	93/72, 85/75,85/81, 85/75,86/82	72/ 72
Avg. Def. (cycl)	16/ 10	24/23, 28/23	33/31, 30/29	18/15	15/4, 19/8	21/ 21	36/32	22/13, 13/2,15/12, 13/2,15/12	24/ 24
Worst Def. (cycl)	32/ 25	47/46, 54/49	65/63, 60/58	35/30	28/14, 37/24	42/ 41	49/45	42/33, 25/10,29/26, 25/10,29/26	48/ 47
RMCS Resp. Red.%	88/ 92	81/82, 78/82	74/76, 77/77	86/90	88/97, 85/94	84/ 84	72/75	83/90,90/98, 88/91, 90/98,88/91	81/ 81
CCS Resp. Red.%	84/ 78	50/48, 47/38	41/40, 53/52	69/63	66/57, 66/53	66/ 66	59/61	75/55,73/68, 71/68, 73/68,73/70	50/ 50

Table 4.2: Characteristics and response reductions with aggressive compiler optimizations for 64-entry register file

	ej	lu	tri	mmul	2d-dct	adp	g721	sha	sus
Live Regs. SP/SB	3/ 16	27/28, 20/38	29/33, 20/24	15/22	20/42, 18/40	15/ 15	{3/5,7/7 8/8,9/9, 12/14}	6/16,11/20, 24/24 11/18,10/13	21/ 26
CCS State Red.%	98/ 89	81/81, 86/74	80/77, 86/83	90/85	86/71, 88/72	90/ 90	{98/97, 95/95,94/94, 94/94,92/90}	96/89, 92/86,83/83, 92/88,93/91	85/ 82
Avg. Def. (cycl)	15/ 3	28/27, 28/24	40/36, 38/36	22/18	16/8, 20/3	24/ 24	49/45	29/23, 28/9,30/11, 28/11,30/6	24/ 24
Worst Def. (cycl)	28/ 11	55/53, 55/51	78/74, 75/73	42/39	30/22, 38/15	48/ 47	63/59	56/51, 54/31,59/26, 54/35,59/26	49/ 44
RMCS Resp. Red.%	88/ 98	79/79, 78/81	69/72, 70/72	83/86	88/94, 84/98	81/ 81	62/65	77/82,78/93, 77/95, 78/91,77/95	80/ 80
CCS Resp. Red%	86/ 83	57/56, 63/51	46/53, 55/53	71/67	72/59, 70/63	70/ 70	56/59	73/67,70/72, 58/71, 70/72,69/72	64/ 60

Table 4.3: Characteristics and response reductions with aggressive compiler optimizations for 128-entry register file

	ej	lu	tri	mmul	2d-dct	adp	g721	sha	sus
Live Regs. SP/SB	3/ 14	18/20, 18/19	13/24, 10/19	9/13	13/15, 11/13	10/ 10	{3/5,7/7 8/8,9/9, 12/14}	6/6,5/9, 10/14, 10/15,9/13	20/ 20
CCS State Red.%	96/ 81	75/72, 75/74	82/67, 86/74	88/82	82/79, 85/82	86/ 86	{96/93, 90/90,89/89, 88/88,83/81}	92/92, 93/88,86/81, 86/79,88/82	72/ 72
Avg. Def. (cycl)	18/ 1	3/1, 3/1	8/1, 8/1	4/2	3/1, 3/1	14/ 14	40/36	2/2, 5/2,4/2, 4/2,4/2	14/ 13
Worst Def. (cycl)	34/ 7	5/3, 5/3	15/5, 15/5	7/4	5/3, 5/3	28/ 27	57/53	4/3, 9/6,7/4, 7/4,7/4	27/ 25
RMCS Resp. Red.%	86/ 99	98/99, 98/99	94/99, 94/99	97/98	98/99, 98/99	89/ 89	69/72	98/98,96/98, 97/98, 97/98,97/98	89/ 90
CCS Resp. Red.%	81/ 75	70/67, 70/69	73/60, 78/68	83/78	77/75, 80/78	73/ 73	56/58	89/89,88/84, 81/77, 81/75,83/78	58/ 59

Table 4.4: Characteristics and response reductions with scalar-only compiler optimizations; 64-entry register file

	ej	lu	tri	mmul	2d-dct	adp	g721	sha	sus
Live Regs. SP/SB	4/ 17	18/20, 18/19	13/24, 10/20	9/13	13/15, 11/13	11/ 11	{3/5,7/7 8/8,9/9, 12/14}	8/8,5/9, 10/15, 10/14,9/13	20/ 20
CCS State Red.%	97/ 88	88/86, 88/87	91/83, 93/86	94/91	91/90, 92/91	92/ 92	{98/97, 95/95,94/94, 94/94,92/90}	94/94, 97/94,93/90, 93/89,94/91	86/ 86
Avg. Def. (cycl)	17/ 1	3/1, 3/1	8/1, 8/1	4/2	3/1, 3/1	14/ 14	54/49	2/1, 5/1,5/1, 5/1,5/1	11/ 11
Worst Def. (cycl)	33/ 7	5/3, 5/3	15/5, 15/5	7/4	5/3, 5/3	28/ 27	71/67	3/2, 10/3,10/4, 9/4,9/4	21/ 20
RMCS Resp. Red%	87/ 99	98/99, 98/99	94/99, 94/99	97/98	98/99, 98/99	89/ 89	58/62	98/99,96/99, 96/99, 96/99,96/99	91/ 91
CCS Resp. Red.%	84/ 84	84/83, 84/84	84/79, 86/82	90/88	88/87, 89/88	80/ 80	52/55	92/92,92/91, 88/88, 88/86,89/88	76/ 76

Table 4.5: Characteristics and response reductions with scalar-only compiler optimizations; 128-entry register file

loop unrolling coupled with instruction scheduling. The scalar-only optimizations include no loop unrolling but all the basic optimizations including scheduling. The second row reports the number of live registers at the switch-points and the switch-blocks for all the benchmarks. This data is represented as a pair of number, the first corresponding to a switch-point and the second to a switch-block. It is noteworthy that this number directly corresponds to the number of execution cycles taken by the custom switch routines. For the single-cluster machine of 64 registers we have assumed one load/store unit, while for the dual-cluster machine with 128 registers two load/store units are assumed. Therefore, for the single-cluster organization, the number of cycles taken by the switch routines is identical to the number of live registers, while for the double-cluster, the number of cycles is equal to the half of the number of live registers. The next row reports the reduction in state (in percentage) that is achieved by storing and restoring the live registers only by the CCS methodology. Similarly, the data for switch-points and switch-blocks is shown for all the benchmarks and their application hotspots. The next pair of rows reports the average and worst preemption deferrals (in cycles) for both switch-points and switch-blocks. These numbers represent the number of cycles from the moment a preemption interrupt occurs to the moment of reaching the switch-point/block. The deferral delay is independent from the type of technique used to preserve the context (CCS or RMCS). Clearly, the deferral is smaller for switch-blocks as they are more quickly reachable on average than switch-points. The last two rows in the tables report the reductions (in percentage) achieved by the RMCS and the CCS methodologies. The baseline architecture is the general-purpose mechanism

		A2	B2	A3	B3	A4	B4
25%	1-cluster	0/0	0/0	1/2	0/2	3/5	3/5
	2-clusters	0/0	0/0	0/4	0/3	3/7	2/7
50%	1-cluster	0/0	0/0	0/0	0/0	1/3	1/3
	2-clusters	0/0	0/0	0/0	0/0	0/3	0/3
75%	1-cluster	0/0	0/0	0/0	0/0	0/1	0/1
	2-clusters	0/0	0/0	0/0	0/0	0/0	0/0

Table 4.6: Live pages exceeding page pool; Aggressive optimizations

for storing the entire register file of the preempted task and loading the register file associated with the preempting task. For the CCS technique, this reduction takes into account the execution cycles needed by the custom switch routines, which, as explained above, are directly proportional to the number of live register for that switch-point/block. For the RMCS technique, a single-cycle switch to remap the appropriate RFPs is assumed.

It is noteworthy, that not only are the two preemption mechanisms significantly faster but also useful instructions are being executed from the preempted task during the deferral interval. This is in contrast with the general-purpose mechanisms where it is not only significantly slower, but is also the case that the delay is pure overhead as only system software instructions are being executed.

As is evident from the tables, when more issue bandwidth is available with the dual-cluster machine, the compiler is more aggressive in unrolling the loops in

		A2	B2	A3	B3	A4	B4
25%	1-cluster	0/0	0/0	0/2	0/0	2/4	3/3
	2-clusters	0/0	0/0	0/1	0/0	0/4	1/1
50%	1-cluster	0/0	0/0	0/0	0/0	0/2	1/1
	2-clusters	0/0	0/0	0/0	0/0	0/0	0/0
75%	1-cluster	0/0	0/0	0/0	0/0	0/0	0/0
	2-clusters	0/0	0/0	0/0	0/0	0/0	0/0

Table 4.7: Live pages exceeding page pool, Scalar-only optimizations

order to uncover and schedule parallel instructions. The loop body increases due to the larger unrolling factor, which in turn leads to increase in the number of live registers throughout the loop body. However, for the minimal points/blocks, the number of live registers is only slightly increased and not doubled due to their typical occurrence towards the beginning or the end of the loop body. Nonetheless, this increase is limited by the available ILP. Furthermore, due to the more load/store units available for this architecture, the actual reductions achieved by the CCS technique are higher compared to the single-cluster architecture; the slight increase in live registers at the switch-points/blocks is compensated by the ability to save and load pairs of registers simultaneously.

In order to evaluate the impact of the size of the pool of extra registers required by the RMCS technique, we have formed sets of multiple tasks for parallel execution. Tables 4.6 and 4.7 show the relation of spare registers pool size and the supported

multiple task set for worst case scenarios - when for all the tasks the preemption occurs within the switch-points/blocks with largest number of live register pages. If all these live pages cannot be accommodated within the extra pool of register pages, the number of pages which exceed the pool must be saved and restored at context switch. Only the non-time critical tasks can be handled in this way, while the rest of the tasks can use the pool of register pages and benefit from the proposed methodology. In Tables 4.6 and 4.7 we show for each task set the number of live register pages, which exceed the pool. We have evaluated three cases for the size of the pool of extra registers: 25%, 50%, and 75% of the size of the baseline register file.

The first row in the tables shows the set name. Set A_i consists of the first i tasks from the computational kernels group; A2={EJ,LU}, A3={EJ, LU, TRI}, A4={EJ, LU, TRI, MMUL}. Sets B_i similarly cover the group of application benchmarks: B2={2D-DCT, ADPCM}, B3={2D-DCT, ADPCM, SHA}, B4={2D-DCT, ADPCM, SHA, SUSAN}. Table 4.6 reports the obtained data for aggressive compiler optimizations. The three main rows contain the data for the cases of 25%, 50%, and 75% extra register in the pool of spare register pages in addition to the baseline register file. For both single- and dual-cluster machines we report the number of live register pages that need to be saved/restored to memory because of insufficient registers in the extra register pool. The two reported numbers correspond to the cases of using switch-points and switch-blocks, respectively. It can be seen from this data that a 25% extra register pool completely covers all the 2-task sets and some of the 3-task sets, while 50% pool completely covers all the 2-, 3-, and some of the

4-task sets. Even in the cases where not all the live pages can be covered by the RMCS technique, the critical tasks can be assigned to the spare pages and utilize the RMCS methodology, while for the remaining non-critical tasks, the context-switch time can still be reduced as only the registers in the live pages would be preserved and restored during task preemption.

Chapter 5

Compiler-driven Register Re-Assignment for Temperature Control

5.1 Introduction

As feature sizes continue scaling down and clock frequencies steadily increase, the *power density* in many modern semiconductor circuits, such as embedded systems and processors, has been doubling every three years and this rate is expected to increase even more [121]. Most prominently, the increased power density has resulted in highly increased temperatures. These new thermal characteristics have, in turn, resulted in serious design challenges, such as reliability problems [122], packaging and cooling costs [123], and elevated leakage power [124].

The reliability of an electronic circuit is exponentially dependent on the junction temperature. A mere $10 - 15^{o}C$ rise in the operating temperature could result in two times reduction in the life span of the circuit. The packaging and cooling solutions designed to remove the heat from the silicon die surface have been typically targeted for the worst case peak temperature. However, due to packaging cost becoming extremely expensive with the increasingly rising temperatures - approximately $10 per Watt after $65^{o}C$ [123], new design strategies have emerged where *thermal management* techniques are used to keep the chip temperature within the thermal capacity of the cooling package and, thus, prevent both transient and permanent system failures.

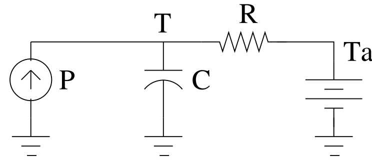


Figure 5.1: Thermal RC model

Leakage power, which is expected to constitute over 50% of the total power consumption in post 90nm CMOS technologies has strong dependency on temperature [124]. The positive feedback loop between leakage power and temperature causes the temperature to continuously rise and, thus, damage the chip when it reaches a certain threshold (runaway temperature). Even if the temperature is controlled below the runaway threshold, the high static power consumption could have a significant impact on many low-power devices operating on battery power.

Due to its high utilization and relatively small area, the register file has been shown to have the highest peak temperature in several studies [125]. In VLIW processors, where multiple instructions are packaged into a single large instruction packet, the register file access frequencies are significantly elevated, which causes the rise of the power density and, thus, peak temperatures. Consequently, reducing the register file power density spots would lead to reduction in peak temperatures for both the entire chip and the register file, which in turn would result in improved reliability and reduced leakage power.

There exists a well-known duality between the power-temperature relation and the current-voltage relation in an RC circuit network. Heat flow can be described as a "current" passing through a thermal resistance, leading to a temperature difference

("voltage difference") between a neighboring point temperate and the temperature of the thermal resistance. Figure 5.1 shows an example of a simplified chip thermal model when only the vertical heat transfer path is considered, which has been shown to be the majority of the heat transfer. The temperature response at time t for a given power dissipation can be obtained through the following equation:

$$T(t) = P(T) \cdot R_{th} + T_a + (T_{init} - P(T) \cdot R_{th} - T_a) \cdot e^{-\frac{t}{R_{th}C_{th}}} \tag{5.1}$$

where $T(t)$ is the transient temperature, $P(T)$ is the total power consumption at temperature T,T_a is the ambient temperature, T_{init} is the initial temperature, R_{th} is the thermal resistance which includes junction-to-ambient thermal resistance of the silicon substrate and the package, and C_{th} is the thermal capacitance. The steady state temperature is modeled as:

$$T_{steady} = P(T_{steady}) \cdot R_{th} + T_a \tag{5.2}$$

where $P(T_{steady})$ is the total power dissipation at T_{steady}. Equation 5.2 can be extended into 5.3,

$$T_{steady} = \frac{P(T_{steady})}{A_{chip}} \cdot r_{A0} + P_{total}(T_{steady}) \cdot R_{heat_sink} + T_a \tag{5.3}$$

where R_{heat_sink} is the thermal resistance of the thermal sink and is less affected by the chip area since the heat is usually spread out more uniformly by the heat spreader before it reaches the sink. Also the heat sink resistance have to be designed to be small enough so that the thermal paste resistance dominates the total thermal

resistance. r_{A0} is the unit area thermal resistance of thermal paste and is fixed. $\frac{P(T)}{A_{chip}}$ represent the power density of the chip and is determined by the ratio of total power dissipation and the chip area. Consequently, reducing the power density of a given block results in effectively reducing the steady temperature for that block.

The proposed temperature reduction methodology introduces a temperature-aware compiler register re-allocation based on application-specific information regarding basic block register accesses and frequency of execution. The register file peak temperature, the hottest spot in any modern embedded processor, is reduced by minimizing the peak power density across the register file. If the register file comprises of multiple banks, the proposed approach distributes the accesses and, hence, the power density uniformly across all the banks. If a monolithic implementation is followed, our technique distributes the accesses uniformly throughout the register file and minimizes the power density and peak temperature. In this case, the proposed algorithm logically partitions the register file into equally sized small partitions and uniformly distributes the accesses among them. From this point on we will use the term "*partition*" to refer to a group of physically adjacent registers, which are not necessary implemented as a separate physical bank but are rather treated by our methodology as a unit entity within the register file.

The power density reduction is achieved through both *register name reassignments* and *register live range reassignments*. These tools are used with the objective of uniformly distributing (spatially) the register accesses across the entire register file. Since the introduced technique only reallocates live ranges and swaps register names, *no performance overhead* is incurred. Moreover, as these transformation are

applied during compile time, *no hardware support* is required, thus no area or power overheads are incurred as well.

5.2 Related Work

Investigating processor thermal characteristics from architecture and systems point of view have been the goal of several research groups and projects in recent years [126]. In [127], the authors have proposed several *Dynamic Thermal Management (DTM)* techniques including DVFS, decoder and i-cache throttling, as well as speculation control. A register relabeling technique is introduced, developed, and evaluated in [128, 129, 130]. The technique relabels the register names with the objective of minimizing dynamic power on the I-Cache data bus and address decoders. In [131] the authors have proposed techniques for reducing cache temperature through power density minimization. A cache block permutation techniques is introduced, which aims at maximizing the distance between blocks with consecutive addresses. Register assignments algorithm for low-power VLIW register files are introduced in [132]. The compile-time reassignment algorithm together with hardware support for dynamically disabling register file sub-banks are used to minimize the dynamic power of large clustered register files. In [133] techniques are proposed to minimize the thermal emergencies in NOC-based systems through compiler-directed power density reduction. Several thermal managing techniques for multicore architectures are explored in [134]. Various core throttling policies applied at core and processor level are explored for chip thermal management. The authors in [135] have

introduced a power density minimization through computational activity migration, while in [136], the authors investigate register temperature reduction through register file and register duplication. At time intervals the hardware is controlled to switch to the other available register file bank and, thus control the peak temperature. In [137], the authors explore the temperature dependence on leakage power of on-chip caches by systematically evaluating and simulating the interdependence between temperature and cache organization and its leakage power. In [138] the author have proposed and evaluated a systematic approach for optimally allocating and placing thermal sensors within a microprocessor. The placement algorithm provides for accurate thermal measurements to be used for run-time temperature optimizations. The relationship between task scheduling and run-time thermal behavior has been investigated in [139]. High-temperature profile tasks are interleaved with low-temperature tasks in order to dynamically control the chip temperature.

The methodology in this chapter aims at reducing the peak temperature of the register file by spreading the register activity and, thus, minimizing the register file power density. Since the register file peak temperature determines the peak temperature for the entire processor, our approach reduces the maximum processor temperature and relaxes the packaging and cooling requirements. Additionally, it also results in improved reliability and reduced leakage power of the register file and its surrounding hardware blocks. All this is achieved with *no impact on performance* and *no additional hardware.*

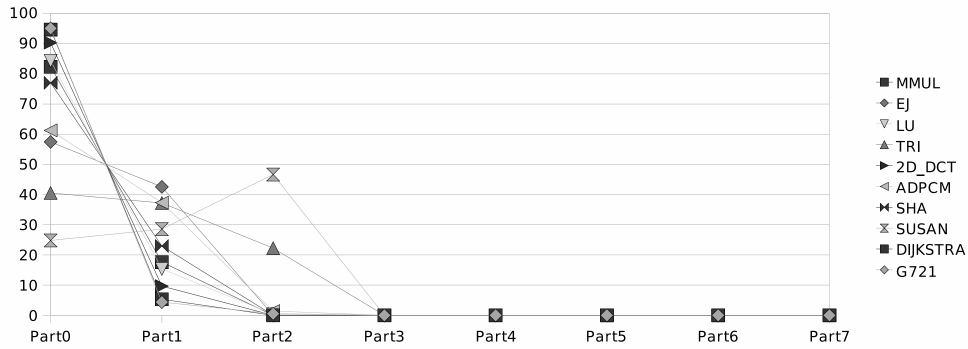


Figure 5.2: Histogram of register accesses; Scalar compiler optimizations

5.3 Motivation for Temperature-Aware Register Re-Allocation

A traditional register allocation scheme in temperature unaware compiler starts by assuming infinite virtual registers for representation and machine-independent optimization purposes. Each virtual register is used to carry a particular value from the moment of its definition (writing into it) to the moment of its last usage before being re-defined. Subsequently, these virtual registers are mapped into the fixed number of available physical registers. To resolve cases where there are not enough physical registers, registers are spilled into memory through load/store instructions. In effect, the load/store instructions terminate and subsequently resume the live range associated to a particular virtual register in order to remain within the limit of available physical registers. The physical registers are usually selected and mapped to a live range following the linear order of the names of currently available registers.

The objective of the traditional register allocation approaches is to minimize the number of load/store instructions introduced for spilling and filling registers.

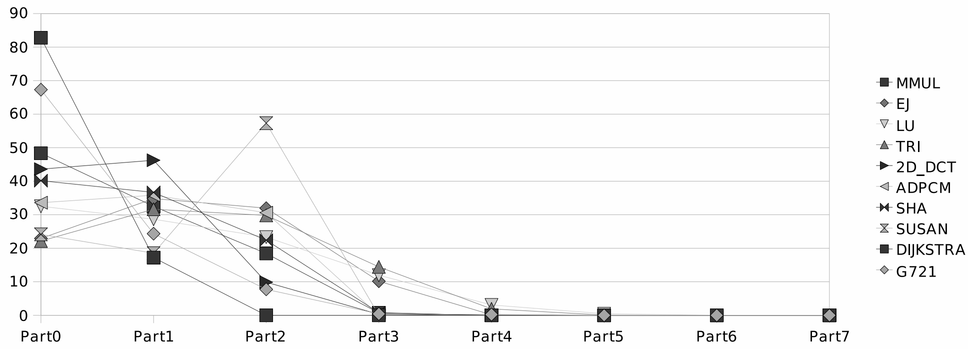


Figure 5.3: Histogram of register accesses; Aggressive compiler optimizations

When the register pressure is high, most of the physical registers are allocated and have to be moved back and forth from memory. In cases (basic blocks, or other code fragments) with low register pressure, only a very few physical registers are allocated and as such contain useful values. The decision regarding which physical registers in particular to be allocated, however, does not take into consideration the distribution of the register accesses. Consequently, often times a significant imbalance in the distribution of the register access activity exists.

Figures 5.2 and 5.3 show the register access distribution for a set of application benchmarks executed on a state-of-the-art embedded multimedia VLIW processor as modeled by the VEX infrastructure [119] from HP labs. The 64-entry register file is logically split into 8 partitions each consisting of 8 registers. The figures report the percentage of the total register accesses for each bank. Figure 5.2 shows the access distribution when the benchmarks have been compiled with fairly traditional optimizations - all scalar optimizations, instruction scheduling and minimal loop unrolling, while Figure 5.3 reports the same data but for the case for very aggressive

compiler optimizations - aggressive loop unrolling combined with trace scheduling. In our experimental study, we have used benchmarks from MediaBench [103] and MiBench[104] such as 2D-DCT (discrete cosine transform widely used in many image and video processing applications), ADPCM and G721 (speech compression codecs), SHA (commonly used hashing algorithm in security), SUSAN (image manipulation), DIJKSTRA (graph optimization). We have also evaluated several numerical computation programs, such as Matrix Multiplication (MMUL), Extrapolated Jacoby (EJ) method, LU decomposition, and TRI (converting a matrix into a triangular form).

It is evident from the above two figures that the first few partitions (mostly the first two) of the physical register file are accessed overwhelmingly many more times than the other partitions. As the power per single register access is identical for all the registers, the power density of the first partition is significantly higher than the power density of the other partitions. The peak temperature of the whole register thus will be reached in the first partition, which will also become significantly hotter than the other parts of the register file. If, on the other hand, the register accesses required by the program and established by the register allocator are evenly distributed through the register file, the power will be uniformly distributed as well and no extreme temperature hotspots will be formed. Fundamentally, this is the objective of the proposed compile-time register re-allocation methodology.

Given a fixed amount of register activity, the lowest power density occurs when the access activities for all the partitions are identical. By rearranging the physical register allocation evenly into multiple partitions, the peak power density and hence

the peak temperature could be reduced. The magnitude of the reduction depends on whether the initial distribution which has resulted from the general-purpose register allocation algorithm is significantly imbalanced.

Since the proposed technique involves register name and live range reassignments only, it can be independently applied on different application hot-spots, which correspond to heavily executed program phases. In this way, each such execution phase that iteratively executes a relatively short code sequence can be optimized independently[1]. Such an approach ensures that the quality of the proposed solution is not diluted throughout a large application program with different execution phases. Furthermore, due to the iterative nature of the execution phases and their relatively short static code size, this approach ensures that the register file is not only accessed uniformly as a whole throughout the program, i.e. spatially, but also that this *uniformity of access is preserved within short time intervals*, i.e. temporally, as well.

Fundamentally, the proposed algorithm can be integrated as a part of the register allocation compiler phase. The selection of an available register to map a particular live range would be driven by the optimization criteria of the proposed technique. The additional profile-based information regarding basic block execution frequency can be provided as a feedback information to the compiler. For the

[1]In this case, a register transfer instruction may have to be introduced in between the application hot-spots for each global variable carried by a register across the program hot-spots. Since such a register transfer, if existing at all, can happen only outside program hot-spots, its impact on performance is practically zero.

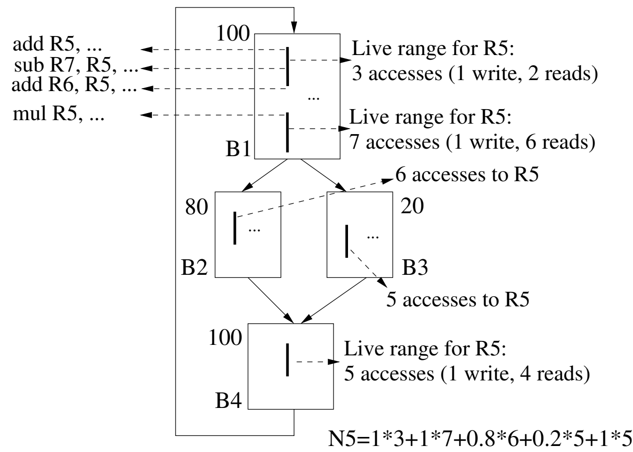


Figure 5.4: A loop Control Flow Graph (CFG) and Register live ranges

purpose of clarity, however, in presenting the algorithm in the subsequent sections of this chapter we assume that it is applied in a compiler pass after the traditional register allocation and after profiling the program under compilation.

The proposed algorithm, which utilizes register name and register live range reassignments does not introduce any performance overhead, because it does not introduce any new instructions within the frequently executed application hotspots. It only judiciously reassigns both the *register names* and the individual *live ranges* carried by the registers to carry each incarnation of the program variables in their define and use cycles. Consequently, the achieved power density and temperature reductions are achieved at *no performance* and *no hardware costs*.

5.4 Temperature-Aware Register Re-Allocation Fundamentals

The fundamental goal of the register allocators is to minimize the amount of memory traffic by mapping as many as possible variables to registers. The proposed

register reassignment for power density minimization is performed after the traditional register allocation phase. Registers are re-allocated without introducing new memory transfers; in essence, our technique exploits the available degrees of freedom that exist when selecting a register name to hold a particular live range.

Figure 5.3 shows an example *Control-Flow Graph (CFG)* for a simple loop with four basic blocks. The numbers associated with each basic block correspond to the percentage of loop execution time spent in the basic block. Such execution frequency numbers can be obtained through profiling. Since the register reallocation is performed independently on the different application hotspots, corresponding to frequently executed loops and functions, the profile-based information regarding register utilization is extremely stable. For instance, our experiments show that for the *G721* speech coder ran on two completely different input data sets, the register access weights (the ratio of register accesses of a particular register to the total number of register accesses) are stable with a worst-case difference of 0.00103. Such a small fluctuation would have no impact whatsoever on the final results achieved by the proposed algorithm.

Subsequently, a live analysis is performed after the traditional compiler register allocation to determine the register live ranges (in case such information is not directly accessible from the already executed register allocator). A register live range is defined as the interval of time (or sequence of instructions) starting from the instruction that writes into the register a new value and ending at the instruction that reads for the last time from it before the register is written to again. Each basic block is composed of the live ranges for all the registers. In this representation

each register has its own column, while the rows correspond to the linear sequence of instructions. In this way the live ranges can be represented as vertical lines. A particular live range can reside within a basic block or can span multiple basic blocks. For each live range, the register is accessed by one write (the first instruction) and one or more reads. In Figure 5.3 we have shown the live ranges for R_5 and have also shown the way we compute the number of accesses to R_5, which we denote with N_5. The number of accesses to R_5 is essentially defined as the weighted sum of the number of accesses for each live range of R_5 with weights equal to the execution frequency for the basic block. In general, the number of accesses to any register for a particular CFG, which represents an application hotspot (function or a loop) is defined by the following equation:

$$N_i = \sum_{b=1}^{nb} W_b (\sum_{j \in L_i(b)} NL_i(j)) \quad (5.4)$$

where N_i is the number of accesses to register R_i, nb is total number of basic blocks, W_b is the weight of block b. $L_i(b)$ is the set of all the live ranges of register i residing in basic block b, and $NL_i(j)$ is the access number of register R_i in live range j. If a live range crosses basic block boundaries, its number of accesses is split and the corresponding basic block weights are used.

As the register file can be divided into several partitions, the number of accesses for each such partition is defined as the sum of accesses for all the registers from that partition. We refer to this value as the partition *weight*.

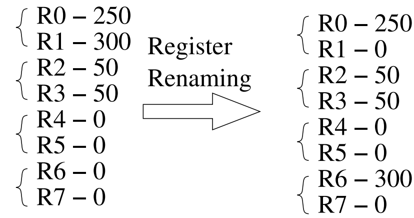


Figure 5.5: Register name reassignment

$$W_i = \sum_{R_j \in Bank(i)} N_j \tag{5.5}$$

Consequently, the power density for a partition with E_B energy per access is:

$$PD_i = \frac{W_i \cdot E_B}{Time \cdot Area} \tag{5.6}$$

Reducing peak temperature is achieved by reducing the power density for all the partitions, which in turn can be achieved by minimizing the maximal weight of a register partition. As the total number of register accesses is a constant, minimizing the maximal partition weight is transformed into uniformly distributing the register accesses to all the partitions. Consequently, the problem is transformed into minimize the $max(W_i)$ given a constant activity $\sum_i W_i$.

5.4.1 Register Name Reassignment (RNR)

A first approach to attack the power density minimization problem, at rather coarse-granularity level, is to employ register name reassignment. If the register name space is permuted and partitioned in such a way, so that the number of accesses to the register file partitions are made uniform, then clearly the register

file power density will be reduced. Consequently, the *Register Name Reassignment (RNR)* problem can be formulated as the procedure of permuting the register name space after register allocation for each application "hot-spot" with the objective of uniformly distributing the total number of register accesses throughout the register file.

The well-known *min-max set-partitioning problem*, which is a known NP-hard problem, can be easily reduced to an instance of this problem - the two problems being almost identical. Such a mapping demonstrates the NP-hardness of our power density minimization problem. It has been shown that the min-max set-partitioning problem is very closely related to the scheduling problem [140]. In order to produce a practical solution we have developed a heuristic algorithm similar to the list-scheduling algorithm. Theoretically the minimum of the $max(W_i)$ is larger than or equal to the average of W_i, where the equality holds only when all the W_i are identical. Consequently, in order to measure the quality of the results produced by the proposed algorithm, we use the ratio of the peak deviation ($max(W_i)$ - $average(W_i)$) to $average(W_i)$, which we denote as PD/AV. For the ideal case when every partition has identical weight, the PD/AV is equal to 0.

In the example shown in Figure 5.5, 8 registers are divided into four partitions. The left side of the figure shows the register activity distribution: the $max(W_i)$ is 550, $PD/AV = 3.385$. The direct and simple way to spread the weight is to exchange R_1 with R_6, and R_3 with R_4. The new distribution is shown on the right side with $PD/AV = 1.846$.

This simple exchange of register names across partitions can be easily imple-

mented by just reassigning their names and updating the instructions accordingly. There is no performance overhead in doing this since no new instructions are introduced. Each such interchange of registers R_i with R_j results in a weight reduction of $N_i - N_j$ for the partition of R_i. Clearly, the granularity of the updates to the register partition weights is within the range of a number of accesses to a single register for the entire CFG under considerations. The set of register access numbers can be substituted for the input set of integers for the set-partitioning problem. The set of integers corresponding to register accesses per partition needs to be partitioned into m groups, such that the sum of integers in each group is equal to $\sum_i W_i/m$, where m is the number of partitions or subsets of the register file. For this step of our approach we offer a heuristic, which is similar to the *list scheduling* heuristic [141] for m-partitioning. Clearly, the granularity of update is a single physical register, which is reassigned to another physical register.

5.4.2 Live Range Reassignment (LRR)

The RNR granularity of adjusting the partition weights, however, can be too large for some programs. Nonetheless, the nature of our problem enables us to significantly reduce the granularity level of the adjustments. As explained above, each register is used to "carry" a set of live ranges mapped to it for the particular CFG. Renaming registers can be thought of as interchanging the entire set of live ranges with the set of live ranges for another register. In many cases, however, it is possible to *reassign only a subset of the live ranges* for some registers instead of

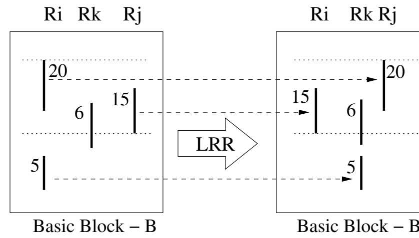


Figure 5.6: Live range reassignment (LRR) procedure

the entire set of live ranges. Such a situation is illustrated in Figure 5.6. The figure shows the register utilization inside a basic block with the live ranges for two of the registers. The representation is identical to the one used in Figure 5.3. The number of register accesses for the live ranges is shown as an integer next to each range.

In this example, between the two dotted lines, the first live range of R_i can be interchanged with the first live range of R_j. This is possible as there is no conflicts (overlaps) with other live ranges of these registers. This particular interchange of live ranges p and q results in a change of $NL_i(p) - NL_j(q)$. The amount is added to the number of accesses of R_i and subtracted from the accesses to R_j. The figure also illustrates a case where a live range (the second of R_i) is moved and reassigned to R_k. With this operation N_k is increased with $NL_i(r)$, while N_i is decreased by the same amount. It is evident that these two transformations are not the same as exchanging register names as they work at much smaller granularity levels - that of a live range. It is also evident that for each of these two operations to be possible, certain conditions must exist. The live range interchange is possible only when it will not introduce overlaps with other live ranges and the live range transfer operation is only possible with a destination register that is “dead” (i.e. contains no live ranges)

for the duration of live range under considerations.

The *Live Range Reassignment (LRR)* procedure effectively utilizes these degrees of freedom to permute the register live ranges carried by all the registers with the similar objective of uniformly distributing the register accesses through the register file. Since the LRR algorithm we propose has an iterative nature, it can be applied as a second step after the RNR procedure. The LRR phase would start operating on the output of the RNR procedure and would attempt to iteratively improve the solution. An alternative approach that we evaluate in our experiments is the application of the LRR procedure as a stand-alone algorithm for power-density minimization. In this case, the iterative procedure starts with the original register assignment as provided by the register allocator.

The proposed LRR procedure iteratively attempts at performing the aforementioned two transformations from the partition with maximal weight to the other partitions. When possible, such transformations are performed only if they will not increase the weight of the destination partition above certain threshold, which is below the current maximal partition. In this way, the iterative process is guaranteed to terminate as the maximal partition weight monotonically decreases and it is bound with the optimal solution of ideal uniform distribution of register access across all the partitions.

Figure 5.7 depicts two alternatives in implementing the proposed register reassignment technique. The first alternative is to use the RNR procedure as a first step in producing a relatively good solution in terms of uniformity of access. Subsequently, this “rough” solution is used as a starting point for the LRR procedure.

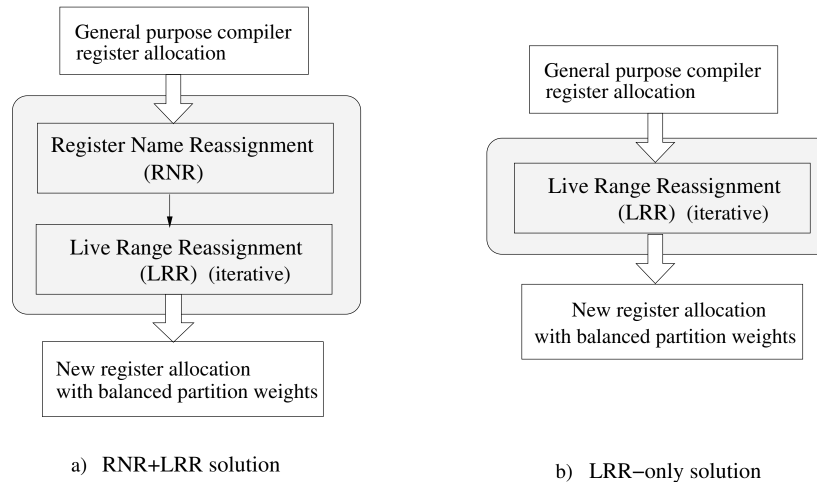
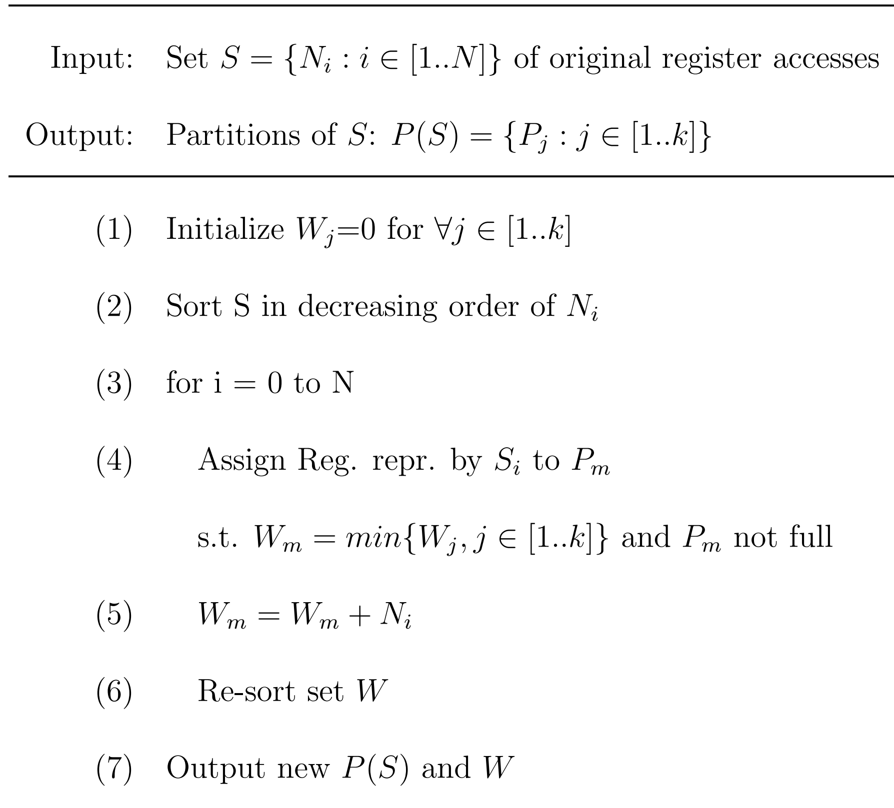


Figure 5.7: Overall algorithms for temperature-aware register reallocation

Because of its iterative nature and finer-granularity in distributing the register accesses, the LRR procedure can be efficiently exploited in improving the solution provided by the RNR step. Clearly, in such RNR+LRR approach, the second phase (LRR) is only used to improve or "fine-tune" the solution of the first phase. An alternative is to start with and execute only the LRR procedure. In this way, the LRR procedure can have the opportunity to uncover a better global solution as compared to the first option when it is only used to improve the RNR solution. Of course, it is impossible to conclude that any of this two approaches is better than the other because of their heuristic nature. It may well be the case, as our experimental results show, that for different programs each of the two approaches could produce slightly better solution. In order to provide the best available solution, a third approach may involve the execution of both sequences (RNR+LRR, and LRR-only) and then chose the better solution.



Input: Set $S = \{N_i : i \in [1..N]\}$ of original register accesses

Output: Partitions of S: $P(S) = \{P_j : j \in [1..k]\}$

(1) Initialize W_j=0 for $\forall j \in [1..k]$

(2) Sort S in decreasing order of N_i

(3) for i = 0 to N

(4) Assign Reg. repr. by S_i to P_m

s.t. $W_m = min\{W_j, j \in [1..k]\}$ and P_m not full

(5) $W_m = W_m + N_i$

(6) Re-sort set W

(7) Output new $P(S)$ and W

Figure 5.8: Pseudo-code for RNR set-partition heuristic

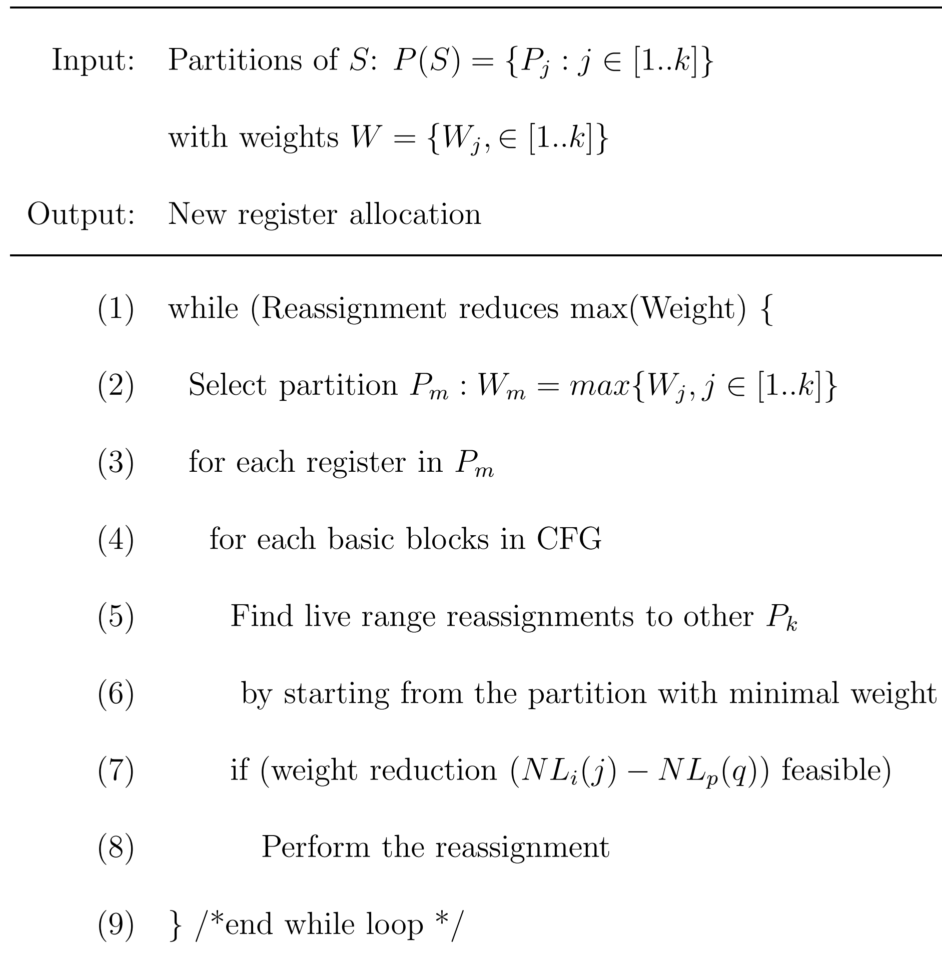
5.5 Register Re-Allocation Algorithms

5.5.1 Set-Partition Heuristic for RNR

The Register Name Reassignment (RNR) procedure involves the reassignment of register names with the objective of uniformly distributing the register accesses into the physical banks, or parts of the register file if no physical banking exists. The algorithm we have developed for this step is similar to the List-scheduling heuristic. The objective function is to assign registers into partitions so that the maximum partition weight is minimized. The heuristic algorithm is shown in 5.8 in a pseudo-code format. As explained in the previous section, the RNR procedure permutes

the register names with the objective of evenly distributing the register access rates across all the partitions. The granularity level is a single register and includes all the live ranges assigned to it by the traditional register allocator.

Initially, the set of N_i, representing the total number of accesses for each register, is sorted in decreasing order; all the partition weights are initialized with 0. Subsequently, following the sorted order the set is traversed and each register (associated with the particular N_i) is assigned to the partition, which currently has a minimal weight. The weight of the partition to which the register (its integer access rate) is assigned to is updated accordingly. Assigning a register to a partition can be thought of as assigning the next available register in that partition to the entire set of live ranges mapped to the initial register by the general-purpose register allocator. In this way, the registers with highest access rate are assigned first to the partitions - in the later phase, the register with less utilization are used to "fill-in" the gaps between the partition weights created by the high utilization registers. The worst-case time complexity of this algorithm is $O(n * log(n))$, where n is the number of registers. This complexity comes from the fact that the algorithm requires one sorting step in the beginning (Step 2), which can be implemented using a $O(n * log(n))$ procedure, such as *Heap Sort*. Steps 3-6 constitute a linear traversal of the sorted set with an additional step of re-sorting the set of partition weights. Since the only purpose of the re-sorting Step 6 is to be able to find the partition with minimal weight, a min-Heap structure can be used to maintain the set of weights and access the one with minimal weight and update it. Therefore, this step can be implemented through a min-Heap structure with $O(log(n))$ time complexity. Consequently, Steps



Input: Partitions of S: $P(S) = \{P_j : j \in [1..k]\}$

with weights $W = \{W_j, \in [1..k]\}$

Output: New register allocation

(1) while (Reassignment reduces max(Weight) {

(2) Select partition $P_m : W_m = max\{W_j, j \in [1..k]\}$

(3) for each register in P_m

(4) for each basic blocks in CFG

(5) Find live range reassignments to other P_k

(6) by starting from the partition with minimal weight

(7) if (weight reduction $(NL_i(j) - NL_p(q))$ feasible)

(8) Perform the reassignment

(9) } /*end while loop */

Figure 5.9: Pseudo-code for live range reassignment (LRR) procedure

3-6 exhibit also time complexity of $O(n * log(n))$, which results in a time complexity of $O(n * log(n))$ for the entire RNR algorithm. For all practical purposes this time complexity results in extremely fast running times - for all our benchmarks well below a second.

5.5.2 Live Range Reassignment Algorithm

The live range reassignment procedure in the proposed approach is an iterative technique that aims at finding a new register allocation, while staying within the

optimal point identified by the traditional register allocator. At each step of the LRR procedure a new live range assignment is identified, which is better in terms of uniformity of register accesses. At each step the goal is to reduce the weight of the partition with maximal weight by reassigning live ranges from registers belonging to it to other partitions. The two transformations of *live range exchange* and *live range migration* as described in the previous section and illustrated in Figure 5.6 are used. The pseudo-code of this algorithmic step is shown in Figure 5.9

The exchange targets the partition with maximal weight. The register with maximal access rate is selected from that partition, and all its live ranges are sequentially considered for reassignment to registers from other partitions. As a destination partition the algorithm selects the one with current minimal weight. After selecting the destination partition, the algorithm iteratively visits all its registers by starting with the one with minimal access rate. First, a *live range migration* is attempted as it usually has a more significant impact on the weight balance than the *live range exchange*. Clearly, for a given destination register it may happen that live range migration is not possible because the destination register carries another live range that overlaps with the one considered for migration. In this case, a live range exchange is attempted. It may also happen that the exchange is not possible as one of the ranges may overlap with another live range in the destination register; in such situation the algorithm continues with the next live range from the selected source register. In the case when the reassignment is possible, it first must be checked for feasibility before actually performing it. Such a reassignment is allowed only when the weight reduction $(NL_i(j) - NL_p(q))$ to the maximal partition will not cause the

weight of the destination partition to rise above a certain threshold. This threshold is defined to be lower than the current maximum partition weight. This reassignment is referred to as "feasible" in the pseudo-code. All such exchanges will result in reductions to the maximal weight. The algorithm continues in a similar fashion with the rest of the registers from the partition with maximal weight in decreasing order of their access rate.

It is evident that an application of the procedure corresponding to the body of the *while* loop in the pseudo-code always results in a reduction of the maximal weight or unchanged register allocation. This procedure is iteratively repeated until it cannot produce a better solution. The LRR algorithm is guaranteed to terminate as the current maximal weight always decreases (the feasibility threshold for live range exchange is always slightly smaller than the current maximal weight) and it is bounded by the ideal solution with perfect uniform distribution of the register accesses across all the partitions.

The worst-case time complexity of the body of the *while* loop is $O(m^2 * l_s * l_d)$, where m is the number of registers within a partition, l_s is the number of live ranges within the source partition (with current maximal wight), and l_d is the number of live ranges for all the registers within the destination partition (with current minimal weight). Since these three values are reasonably small for a typical program hot-spot, the running times of the LRR algorithm for our benchmarks were practically unnoticeable and for all of them within a few seconds.

5.6 Experimental results

In our experimental study we have implemented and analyzed the proposed temperature-aware register reallocation algorithms. A quantitative analysis on the proposed technique as compared to traditional register allocation has been performed. We start by performing an application profiling and extraction of the application CFGs. For this we have utilized the VEX package [119], which is developed and provided by HP research labs. It includes a state-of-the-art optimizing VLIW compiler and a compiled-simulator tool chain. The VLIW processor core, which is based on the Lx/ST200 family of processors, can be configured into various architectures with multiple clusters. Each cluster is configured to have an integer register file, four integer ALUs, two 16*32-bit multiply units, and a data cache port. The cluster can issue up to four operations per instructions. The register set for each cluster consists of 64 general-purpose 32-bit registers and 8 1-bit branch registers. For our experiments we have assumed a single cluster architecture with a single 64-entry integer register file. In order to explore the sensitivity of the proposed algorithm on the partition size, we have explored two different cases: 4 partitions and 8 partitions. In the former case, the 64 registers are partitioned into 4 groups each of 16 consecutive registers, while in the latter case into 8 groups of 8 registers.

To determine the register live ranges we have implemented a live analysis tool, which is integrated with our implementation of the proposed register reallocation algorithm. The CFG and register usage and definitions are produced by parsing the generated assembly file. The profile information, as well as the baseline register

access rate and access distributions are obtained through simulations by using the compiled simulation technology of VEX. We have obtained the power characteristics of the register file by modeling it with CACTI 4.2 [109]. The per access power consumption is provided, which we have used to compute the total dynamic power consumed at the register files and partitions. In the last step, in order to determine the temperature of each register partition we have used the *Hotspot* [126] temperature modelling tool. As an input it uses the traces of dynamic power expended at the register file together with a floor-plan of the register file/partitions modules. The power simulation and the temperature estimation phases are executed iteratively until temperature converges to a steady value.

We have performed the control flow and register liveness analysis on the VEX generated assembly files. The register access statistics for all the live ranges inside the basic blocks is collected as well. In order to analyze the impact of aggressive compiler optimizations on the proposed methodology, we have used two compiler setups. The first setup uses optimization options, which include very aggressive loop unrolling and code scheduling. The other option that we have explored includes traditional scalar optimizations but no loop unrolling. We have experimented on 10 benchmarks programs from various application domains, such as speech, audio, image, and numerical/signal processing programs. The set of benchmarks was outlined in Section 5.3.

Table 5.1 shows the thermal characteristic of the baseline case with aggressive compiler optimizations and its associated traditional, temperature-unaware register allocation. In that table we report the results for both 8 and 4 register file parti-

	2D_DCT	*ADPCM*	*SHA*	*SUSAN*	*DIJKSTRA*
AVRG (x1K,4-B)	4327	9639	122254	24128	215249
AVRG (x1K,8-B)	2163	4820	61127	12064	107624
SDev (x1K,4-B)	6519	10996	153430	24640	372822
SDev (x1K,8-B)	3288	6243	81265	18568	233806
PD/AV (4-B/8-B)	2.59/2.7	1.78/1.87	2.07/2.21	1.29/3.59	3/5.62
Peak Temp. (C)	73.05	69.45	76.95	66.85	78.85
	G721	*MMUL*	*EJ*	*LU*	*TRI*
AVRG (x1K,4-B)	1196918	1388	12970	12218	989
AVRG (x1K,8-B)	598459	694	6485	6109	495
SDev (x1K,4-B)	1848463	1842	13246	12217	960
SDev (x1K,8-B)	1061323	978	7364	6290	512
PD/AV (4-B/8-B)	2.67/4.38	2.23/2.87	1.31/1.79	1.45/1.6	1.15/1.53
Peak Temp. (C)	84.85	76.85	78.65	92.25	73.75

Table 5.1: Baseline thermal characteristics; Aggressive compiler optimizations

	2D_DCT	*ADPCM*	*SHA*	*SUSAN*	*DIJKSTRA*
AVRG (x1K,4-B)	5789	10215	167969	24752	278319
AVRG (x1K,8-B)	2895	5107	83984	12376	139160
SDev (x1K,4-B)	10027	17362	290817	24863	482063
SDev (x1K,8-B)	6851	9020	171449	16967	346398
PD/AV (4-B/8-B)	3/6.23	2.94/3.9	3/5.16	1.13/2.73	3/6.58
Peak Temp. (C)	55.25	64.45	65.75	64.95	70.85

	G721	*MMUL*	*EJ*	*LU*	*TRI*
AVRG (x1K,4-B)	1435179	2116	21401	11269	1429
AVRG (x1K,8-B)	717590	1058	10701	5635	715
SDev (x1K,4-B)	2468377	3666	37068	19428	1817
SDev (x1K,8-B)	1794100	2289	18807	12440	964
PD/AV (4-B/8-B)	2.98/6.61	3/5.59	3/3.6	2.99/5.74	2.11/2.24
Peak Temp. (C)	67.65	59.75	83.65	68.85	71.35

Table 5.2: Baseline thermal characteristics; Scalar compiler optimizations

tions. The first two rows report the average number of accesses per partition (in thousands) for the case of 4 and 8 partitions, respectively. The average number of accesses per partition represents the optimal solution for our problem of evenly distributing the register accesses and is used to measure the quality of the solutions produced by our technique. The subsequent two rows in the table report the standard deviation of the register access distribution throughout the 8 and 4 partitions, respectively. The next row, labeled *PD/AV*, reports the peak deviation (the difference between the maximal access number and the average access number) to the average access. This is the value of PD/AV as described in Section 5.4. For the ideal case of perfectly uniform distribution, this ratio is zero. Each cell of this row reports the two values for 8 and 4 partitions, respectively. The last row, labeled *Peak Temp.*, shows the peak (maximal) temperature (in C) selected out of all the register file partitions. This temperature represents the hottest point in the register file and its minimization is the goal of our methodology. Table 5.2 has an identical organization and content, and it reports the baseline temperature characteristics where the application benchmarks are compiled with traditional scalar optimizations only. Minimal loop unrolling and instruction scheduling is performed, while no aggressive trace scheduling is performed. It can be observed that the peak temperatures for the scalar-optimizations are lower. This can be explained by the fact that with scalar-only optimizations, the register file is not as frequently accessed and exhibits a lower baseline temperature. Since the compiler does not aggressively exploit ILP, the number of register accesses per unit time is significantly less than in the case of aggressive loop unrolling and scheduling.

	2D_DCT	*ADPCM*	*SHA*	*SUSAN*	*DIJKSTRA*
SDev (4-B/8-B) x1K	6/57	143/124	86/65	0.4/1579	100/18880
PD/AV (4-B)	0.00086	0.01482	0.00040	0.00001	0.00069
PD/AV (8-B)	0.060	0.010	0.004	0.340	0.110
Peak Temp. (4-B)	68.65	66.15	72.45	64.55	72.25
Peak Temp. (8-B)	67.95	65.45	71.55	64.15	71.65
Reduct. (C) (4-B/8-B)	4.4/5.1	3.3/4	4.5/5.4	2.3/2.7	6.6/7.2
	G721	*MMUL*	*EJ*	*LU*	*TRI*
SDev (4-B/8-B) x1K	511/6812	4/81	115/168	0.007/0.1	0.2/4
PD/AV (4-B)	0.00047	0.00467	0.00920	0.00000	0.00014
PD/AV (8-B)	0.009	0.240	0.010	0.000	0.010
Peak Temp. (4-B)	78.05	72.05	75.25	87.25	70.95
Peak Temp. (8-B)	77.05	71.55	74.25	85.95	70.15
Reduct. (C) (4-B/8-B)	6.8/7.8	4.8/5.3	3.4/4.4	5/6.3	2.8/3.6

Table 5.3: RNR+LRR thermal characteristics; Aggressive compiler optimizations

	2D_DCT	*ADPCM*	*SHA*	*SUSAN*	*DIJKSTRA*
SDev (4-B/8-B) x1K	5/14	142/123	86/65	291/4795	117/18899
PD/AV (4-B)	0.00075	0.01480	0.00040	0.01665	0.00054
PD/AV (8-B)	0.00000	0.01000	0.00040	0.34390	0.10850
Peak Temp. (4-B)	68.55	65.85	72.25	64.25	72.25
Peak Temp. (8-B)	67.85	65.45	71.55	64.35	71.75
Reduct. (C) (4-B/8-B)	4.5/5.2	3.6/4	4.7/5.4	2.6/2.5	6.6/7.1

	G721	*MMUL*	*EJ*	*LU*	*TRI*
SDev (4-B/8-B) x1K	553/6485	4/267	59/124	0.006/4	0.2/2
PD/AV (4-B)	0.00048	0.00523	0.00459	0.00000	0.00013
PD/AV (8-B)	0.00870	0.24000	0.02000	0.00043	0.00000
Peak Temp. (4-B)	77.95	71.85	75.05	86.95	70.85
Peak Temp. (8-B)	77.05	72.05	74.25	85.95	70.15
Reduct. (C) (4-B/8-B)	6.9/7.8	5/4.8	3.6/4.4	5.3/6.3	2.9/3.6

Table 5.4: LRR-only thermal characteristics; Aggressive compiler optimizations

	2D_DCT	*ADPCM*	*SHA*	*SUSAN*	*DIJKSTRA*
SDev x1K (4-B/8-B)	439 /830	143 /189	86 /1154	192 /1655	8855 /30783
PD/AV (4-B)	0.08	0.01	0.0003	0.0078	0.049
PD/AV (8-B)	0.43	0.01	0.02	0.31	0.11
Peak Temp. (4-B)	52.75	60.85	61.55	62.55	65.35
Peak Temp. (8-B)	52.55	60.15	60.95	62.35	65.05
Reduct. (C) (4-B/8-B)	2.5/2.7	3.6/4.3	4.2/4.8	2.4/2.6	5.5/5.8

	G721	*MMUL*	*EJ*	*LU*	*TRI*
SDev x1K (4-B/8-B)	14781 /337669	191 /377	392 /1028	415 /1329	25 /24
PD/AV (4-B)	0.017	0.16	0.03	0.06	0.02
PD/AV (8-B)	0.6	0.41	0.17	0.2	0.02
Peak Temp. (4-B)	63.65	56.65	77.25	64.15	67.55
Peak Temp. (8-B)	63.35	56.55	76.25	63.85	66.65
Reduct. (C) (4-B/8-B)	4/4.3	3.1/3.2	6.4/7.4	4.7/5	3.8/4.7

Table 5.5: RNR+LRR thermal characteristics; Scalar compiler optimizations

	2D_DCT	*ADPCM*	*SHA*	*SUSAN*	*DIJKSTRA*
SDev x1K (4-B/8-B)	452 /6851	143 /9020	86 /171449	5685 /16987	17722 /346398
PD/AV (4-B)	0.07617	0.01	0.0003	0.19247	0.08649
PD/AV (8-B)	0.08000	0.01	0.0100	0.31000	0.11000
Peak Temp. (4-B)	52.25	60.15	61.55	62.35	65.15
Peak Temp. (8-B)	52.05	59.65	60.95	62.65	64.95
Reduct. (C) (4-B/8-B)	3/3.2	4.3/4.8	4.2/4.8	2.6/2.3	5.7/5.9

	G721	*MMUL*	*EJ*	*LU*	*TRI*
SDev x1K (4-B/8-B)	25566 /1794100	391 /2289	376 /188807	1015 /12440	26 /964
PD/AV (4-B)	0.01736	0.16549	0.03007	0.07733	0.01809
PD/AV (8-B)	0.59000	0.41000	0.16000	0.19000	0.02000
Peak Temp. (4-B)	62.75	55.65	76.05	63.05	66.45
Peak Temp. (8-B)	62.85	56.35	75.65	62.95	66.05
Reduct. (C) (4-B/8-B)	4.9/4.8	4.1/3.4	7.6/8	5.8/5.9	4.9/5.3

Table 5.6: LRR-only thermal characteristics; Scalar compiler optimizations

Tables 5.3, 5.4, 5.5, and 5.6 show the power-density and temperature characterstics after the application of the proposed methodology. The four tables have identical structure; the first two tables report the temperature reduction results for the case of aggressive compiler optimizations, while the last two are for the case of scalar optimizations. Tables 5.3 and 5.5 report the results for the RNR+LRR methodology, while the other two are for LRR-only approach. The first row in each table shows the standard deviation (in thousands) of the register access distribution throughout the register partitions. The standard deviations for the case of 4 and 8 partitions are reported in each cell, respectively. The next two rows report the achieved PD/AV ratio for 4 and 8 partitions, respectively. The subsequent two rows show the achieved peak temperatures for 4 and 8 partitions, respectively, while the last row reports the achieved temperature reduction for both cases.

It can be seen from the results that after applying the proposed algorithm, the standard deviation and the PD/AV ratios are significantly reduced and for all of the benchmarks are very close to their optimal achievable value. As a result, the obtained peak temperature reductions are in the range of 4 to 7 degrees Celsius. The achieved reductions are somewhat similar for the two different compiler optimization cases - slightly smaller for the scalar-only optimizations. This follows from the fact that with scalar-only optimizations, the register file is not as frequently accessed and exhibits a lower baseline temperature, as was explained for the baseline results. The achieved results also demonstrate that the temperature reductions for 8 partitions are consistently larger (with up to 1 C degree) than the reductions for 4 partitions. Both the RNR+LRR and LRR-only approaches consistently find better solutions

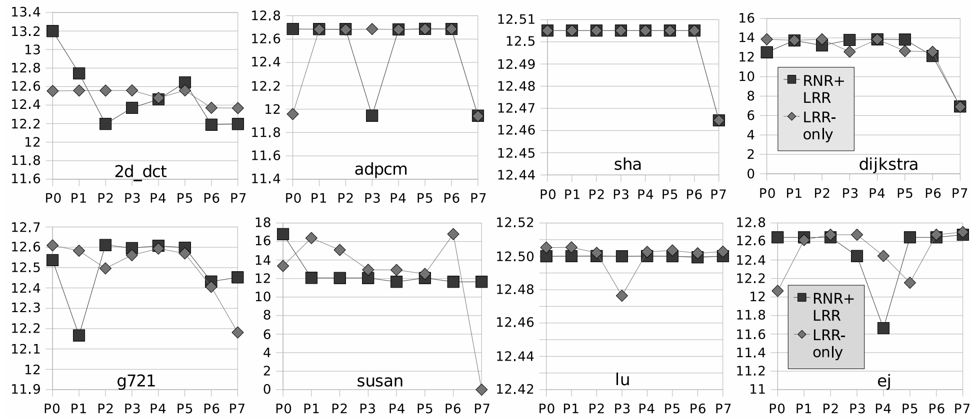


Figure 5.10: Achieved register access distribution; Aggressive compiler optimizations

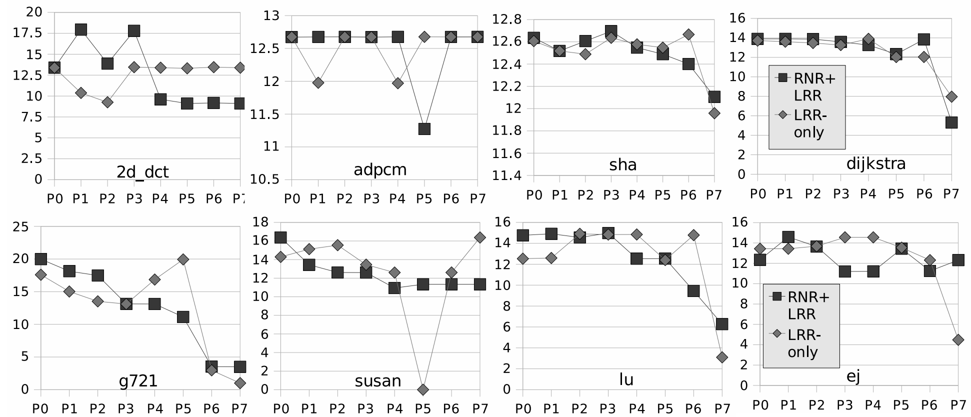


Figure 5.11: Achieved register access distribution; Scalar compiler optimizations

when more partitions are available for distributing the register accesses.

Figures 5.10 and 5.11 depict and compare the register access distribution achieved by the RNR+LRR and the LRR-only approaches. The percentage of utilization for each partition is depicted as a point in the graphs. The graphs visually compare the results of the two methodologies. The figures show results for the case of 8 partitions, since the differences between the two approaches are more prominent in this case. It is evident from the figures that the achieved distribution and the correspondent peak temperature reductions are very close for both approaches with

slightly more stable and uniform solutions provided by the RNR+LRR algorithm. An extreme case where the RNR+LRR stability is evident is the *susan* benchmark. It can be seen that the LRR-only approaches underutilizes one partition for either case of compiler optimizations, while the RNR+LRR sequence provides better solutions in terms of uniformity of access even though the peak densities are almost identical for both approaches.

Chapter 6

Conclusion

As process technologies continue to evolve, the semiconductor integration density would achieve even higher levels in the future. This in turn would result in more complex application systems integration on the SOC platform. The overall system optimization would be performed with multi-dimensions of considerations including performance, power consumption, security, design cost, time-to-market, flexible upgrade and maintenance. Reconfigurable SOC platforms integrated with processors and customized hardwares would continue expanding the market deployments and increasingly drawing the attentions from both industries and academia.

In the cross-layer customization methodology, system-level information including intra-tasking information such as CDFG, intra-task run-time information such as task preemption graphs, and hardware information such as the caches utilization can be exploited all at the same framework. Thus with specific target of optimization objective, the global optimization methods can by utilized. The customization controls are partitioned into different layer and loaded in the platform with the binary source code. During the run-time, all the customized modules can collaborated with each other to fine-tune the system components according to the run-time application properties and system requirement. The overall system performance and power efficiency can be significant improved.

Multi-Processor SOC design has been widely utilized by industry. Task parallelization, data locality and memory bandwidth are all critical considerations of multiple processor systems design. On the other hand, communication and synchronization among tasks introduce new design constraints. The memory systems and inter-processor interconnection strongly affect the performance and power overhead, and dominate the scalability of the whole system. The general purpose mechanisms for preserving cache consistency and task synchronization have significant redundancy. Application knowledge regarding computation and communication patterns, and time of synchronization can be efficiently exploited by a customizable communication architecture so that the associated power and performance overhead are minimized.

In the future researches, more design space exploration and efficient global optimization need to be further investigated. The new era of multi-processor platform (MPSOC) bring new dimensions of complexity. There are more design challenges in the MPSOC platforms such as application mapping, inter-task communication and synchronization, memory hierarchy(cache) coherence management, and interconnection networks configuration. The cross-layer customization could also be extended to cross the processor customization by utilizing potential inter-processor information.

Bibliography

[1] H. Fleisher and L. I. Maissel, “An Introduction to Array Logic”, *IBM J. Res. Develop.*, pp. 98–109, March 1975.

[2] R. A. Wood, “A High Density Programmable Logic Array Chip”, *IEEE Trans. on Computers*, vol. C-28, n. 9, pp. 602–608, September 1979.

[3] Claus Kuhnel, *Avr Risc Microcontroller Handbook*, Elsevier, 1998.

[4] Steve Furber, *ARM System Architecture*, Addison-Wesley Longman, 1998.

[5] IBM, *PowerPC Architecture Book*.

[6] WindRiver, *VxWorks, http://www.windriver.com*, Wind River.

[7] Karim Yaghmour, *Building Embedded Linux Systems*, O’Reilly Media, Inc., 2003.

[8] Edward L. Lamie, *Real-Time Embedded Multithreading : Using ThreadX and ARM*, CMP Books, 2004.

[9] James Y. Wilson and Aspi Havewala, *Building Powerful Platforms with Windows CE*, Addison-Wesley Professional, 2001.

[10] X. Wang and S. Ziavras, “Parallel LU Factorization of Sparse Matrices on FPGA-Based Configurable Computing Engines”, 2004.

[11] X. Zhou and P. Petrov, “Energy-efficient address translation for virtual memory support in low-power and real-time embedded processors”, in *International Conference on Hardware/Software Codesign and System Synthesis*, pp. 33–38, New York, NY, USA, 2005, ACM Press.

[12] X. Zhou and P. Petrov, “Direct Address Translation for Virtual Memory in Energy-Effcient and Real-Time Embedded Systems”, *to appear in ACM Transactions on Embedded Computing Systems (TECS)*.

[13] X. Zhou and P. Petrov, “Arithmetic-based address translation for energy-efficient virtual memory support in low-power, real-time embedded systems”, in *Symposium on Integrated Circuits and Systems Design (SBCCI)*, pp. 86 – 91, September 2005.

[14] X. Zhou and P. Petrov, “Low Power and Real-Time Translation through Arithmetic Operations for Virtual Memory Support in Embedded Systems”, *to appear in IET Computers and Digital Techniques*.

[15] X. Zhou and P. Petrov, “Low Power Cache Organization Through Selective Tag Translation for Embedded Processors with Virtual Memory Support”, in *Great Lakes Symposium on VLSI Systems (GLSVLSI)*, pp. 398 – 403, May 2006.

[16] X. Zhou and P. Petrov, “Heterogeneously Tagged Caches for Low-Power Embedded Systems with Virtual Memory Support”, *under review after first round revision in ACM Transactions on Design Automation of Electronic Systems (TODAES).*

[17] X. Zhou and P. Petrov, “The interval page table: virtual memory support in real-time and memory-constrained embedded systems”, in *Conference on Integrated Circuits and Systems Design (SBCCI)*, pp. 294–299, 2007.

[18] X. Zhou and P. Petrov, “Rapid and Low-Cost Context-Switch through Embedded Processor Customization for Real-Time and Control Applications”, in *Design Automation Conference (DAC)*, pp. 352 – 357, July 2006.

[19] X. Zhou and P. Petrov, “Cross-Layer Customization for Rapid and Low-Cost Task Preemption in Multi-Tasked Embedded Systems”, *under review in ACM Transactions on Embedded Computing Systems (TECS).*

[20] X. Zhou and P. Petrov, “Application-Driven Register File Mapping for Rapid Task Preemption in Real-Time, Multi-Tasked Embedded Systems”, in *Workshop on Application Specific Processors (WASP)*, October 2007.

[21] X. Zhou, C. Yu, A. Dash and P. Petrov, “Application-Aware Snoop Filtering for Low-Power Cache Coherence in Embedded Multiprocessors”, *to appear in ACM Transactions on Design Automation of Electronic Systems (TODAES).*

[22] X. Zhou and P. Petrov, “Compiler-Driven Register Re-Assignment for Register filer Power-Density and Temperature Reduction”, *accepted in Design Automation Conference (DAC) 2008.*

[23] N. Kim, T. Austin, D. Blaauw, T. Mudge, K. Flautner, J. Hu, M. Irwin, M. Kandemir and N. Vijaykrishnan, “Leakage Current - Moore’s Law Meets Static Power”, 2003.

[24] T. Pering, T. Burd and R. Brodersen, “Dynamic Voltage Scaling and the Design of a Low-Power Microprocessor System”, 1998.

[25] T. Simunic, L. Benini, A. Acquavia, P. Glynn and G. De Micheli, “Dynamic voltage scaling and power management for portable systems”, in *38th DAC*, pp. 524–529, June 2001.

[26] Intel, *Intel XeonTM Processor at 1.40 GHz, 1.50 GHz, 1.70 GHz and 2 GHz.*

[27] Transmeta, *Crusoe LongRun Power Management White Paper.*

[28] C. H. Tan and J. Allen, “Minimization of Power in VLSI Circuits Using Transistor Sizing, Input Ordering, and Statistical Power Estimation”, in *Intl Workshop on Low Power Design*, pp. 75–80, April 1994.

[29] S. C. Prasad and K. Roy, "Circuit Optimization for Minimization of Power Consumption Under Delay Constraint", in *Intl Workshop on Low Power Design*, page 1520, April 1994.

[30] Anantha P. Chandrakasan, *Low-Power Digital CMOS Design*, PhD thesis, University of California at Berkeley, 1994.

[31] P. Landman and J. Rabaey, "Black-box capacitance models for architectural power analysis", in *International Workshop on Low Power Design*, page 65170, April 1994.

[32] S. Powell et al., "Estimating power dissipation of VLSI signal processing chips: The PFA technique", in *VLSI Signal Processing*, page 250259, 1990.

[33] S. Manne, A. Klauser and D. Grunwald, "Pipeline gating: speculation control for energy reduction", in *25th ISCA*, pp. 132–141, June 1998.

[34] Ganesh Dasika, "Partitioning Variables across Register Windows to Reduce Spill Code in a Low-Power Processor Rajiv A. Ravindran, Student Member, IEEE, Robert M. Senger, Eric D. Marsman,".

[35] J. Kin, M. Gupta and W. H. Mangione-Smith, "The filter cache: an energy efficient memory structure", in *30th MICRO*, pp. 184–193, April 2001.

[36] A. Ma, M. Zhang and K. Asanovic, "Way memoization to reduce fetch energy in instruction caches", in *Workshop on Complexity-Effective Design, 28th ISCA*, June 2001.

[37] H. Lekatsas, J. Henkel and W. Wolf, "Code compression for low power embedded system design", in *Design Automation Conference*, pp. 294–299, June 2000.

[38] P. P. Sotiriadis and A. Chandrakasan, "Low power bus coding techniques considering inter-wire capacitance", in *Custom Integrated Circuits Conference*, pp. 507–510, 2000.

[39] Chinnakrishnan S. Ballapuram, Hsien-Hsin S. Lee and Milos Prvulovic, "Synonymous address compaction for energy reduction in data TLB", in *ISLPED '05: Proceedings of the 2005 international symposium on Low power electronics and design*, pp. 357–362, New York, NY, USA, 2005, ACM Press.

[40] W. Yuan and K. Nahrstedt, "Integration of dynamic voltage scaling and soft real-time scheduling for open mobile systems", 2002.

[41] P. Petrov, D. Tracy and A. Orailoglu, "Energy-Efficient Physically Tagged Caches for Embedded Processors with Virtual Memory", in *Design Automation Conference*, pp. 17–22, June 2005.

[42] P. R. Panda, F. Catthoor, N. D. Dutt, K. Danckaert, E. Brockmeyer, C. Kulkarni, A. Vandercappelle and P. G. Kjeldsberg, "Data and memory optimization techniques for embedded systems", *ACM Transactions on Design Automation of Electronic Systems*, vol. 6, n. 2, pp. 149–206, 2001.

[43] L. Benini, A. Macii and M. Poncino, "Energy-aware design of embedded memories: A survey of technologies, architectures, and optimization techniques", *ACM Transactions on Embedded Computing Systems*, vol. 2, n. 1, pp. 5–32, 2003.

[44] P. Denning, "Virtual Memory", *ACM Computing Surveys*, vol. 28, n. 1, pp. 213–216, 1996.

[45] B. Jacob and T. Mudge, "Virtual memory: issues of implementation", *IEEE Computer*, vol. 31, n. 6, pp. 33–43, June 1998.

[46] Intel Corporation, *Intel XScale Microarchitecture*.

[47] S. B. Furber, *ARM System-on-Chip Architecture*, Addison-Wesley Publishing Co, Boston, MA, 2000.

[48] ARM Ltd., *ARM920T Technical Reference Manual*.

[49] M. Ekman, F. Dahlgren and P. Stenstrom, "TLB and snoop energy-reduction using virtual caches in low-power chip-microprocessors", in *ISLPED*, pp. 243–246, August 2002.

[50] J. Montanaro et al., "A 160Mhz, 32b 0.5W CMOS RISC Microprocessor", in *IEEE ISCC*, pp. 214–229, February 1996.

[51] T. Juan, T. Lang and J. J. Navarro, "Reducing TLB power requirements", in *ISLPED*, pp. 196–201, August 1997.

[52] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju and G. Chen, "Generating Physical Addresses Directly for Saving Instruction TLB Energy", *International Symposium on Microarchitecture (MICRO)*, page 185, 2002.

[53] X. Qiu and M. Dubois, "Towards virtually-addressed memory hierarchies", in *HPCA*, pp. 51–62, January 2001.

[54] D. Fan, Z. Tang, H. Huang and G. Gao, "An energy efficient TLB design methodology", in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 351–356, August 2005.

[55] C. Ballapuram, H. Lee and M. Prvulovic, "Synonymous address compaction for energy reduction in data TLB", in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 357–362, August 2005.

[56] H. Lee and C. Ballapuram, "Energy efficient D-TLB and data cache using semantic-aware multilateral partitioning", in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 306–311, August 2003.

[57] C. Zhang, "Balanced Cache: Reducing Conflict Misses of Direct-Mapped Caches", in *International Symposium on Computer Architecture (ISCA)*, pp. 155–166, June 2006.

[58] T. Givargis, “Zero Cost Indexing for Improved Embedded Processor Cache Performance”, *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 11, n. 1, pp. 3–25, January 2006.

[59] J. H. Lee, J. S. Lee, S. Jeong and S. Kim, “A banked-promotion TLB for high performance and low power”, in *ICCD*, pp. 118–123, September 2001.

[60] M. Kandemir, I. Kadayif and G. Chen, “Compiler-Directed Code Restructuring for Reducing Data TLB Energy”, in *International Conference on Hardware/Software Codedesign and System Synthesis, CODES+ISSS*, pp. 98–103, September 2004.

[61] M. Simpson, B. Middha and R. Barua, “Segment protection for embedded systems using run-time checks”, in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pp. 66–77, 2005.

[62] B. Calder, C. Krintz, S. John and T. Austin, “Cache-conscious Data Placement”, in *Proceedings of the 8th International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 139–149, October 1998.

[63] L. Benini, F. Menichelli and M. Olivieri, “A Class of Code Compression Schemes for Reducing Power Consumption in Embedded Microprocessor Systems”, *IEEE Transactions on Computers*, vol. 53, n. 4, pp. 467–482, 2004.

[64] T. M. Chilimbi, M. D. Hill and J. R. Larus, “Cache-conscious structure layout”, in *PLDI ’99: Proceedings of the ACM SIGPLAN 1999 conference on Programming language design and implementation*, pp. 1–12, 1999.

[65] C. Kulkarni, C. Ghez, M. Miranda, F. Catthoor and H. De Man, “Cache Conscious Data Layout Organization for Conflict Miss Reduction in Embedded Multimedia Applications”, *IEEE Transactions on Computers*, vol. 54, n. 1, pp. 76–81, 2005.

[66] J. Kim, S. Min, S. Jeon, B. Ahn, D. Jeong and C. Kim, “U-cache: a cost-effective solution to synonym problem”, in *HPCA*, pp. 243–252, January 1995.

[67] P. Petrov, D. Tracy and A. Orailoglu, “Energy-Efficient Physically Tagged Caches for Embedded Processors with Virtual Memory”, in *DAC*, pp. 17–22, June 2005.

[68] I. Kadayif, P. Nath, M. Kandemir and A. Sivasubramaniam, “Compiler-directed physical address generation for reducing dTLB power”, in *International Symposium on Performance Analysis of Systems and Software (ISPASS)*, pp. 161– 168, 2004.

[69] B. Middha, M. Simpson and R. Barua, “MTSS: multi task stack sharing for embedded systems”, in *CASES*, pp. 191–201, New York, NY, USA, 2005.

[70] H. Hansson, L. Lawson, O. Bridal, C. Eriksson, S. Larsson, H. Lon and M. Stromberg, "BASEMENT: an architecture and methodology for distributed automotive real-time systems", *IEEE Transactions on Computers*, vol. 46, n. 9, pp. 1016–1027, September 1997.

[71] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer and D. Culler, "TinyOS: An operating system for wireless sensor networks", *Ambient Intelligence, Springer-Verlag*, 2005.

[72] S. Bhatti, J. Carlson, H. Dai, J. Deng, J. Rose, A. Sheth, B. Shucker, C. Gruenwald, A. Torgerson and R. Han, "MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms", *MONET, Special Issue on Wireless Sensor Networks*, vol. 10, n. 4, pp. 563–579, Auguest 2005.

[73] S. Shivshankar, S. Vangara and A. Dean, "Balancing Register Pressure and Context-Switching Delays in ASTI Systems", in *Proceedings of the 2005 International Conference on Compilers, Architectures and Synthesis for Embedded Systems*, pp. 286–294, September 2005.

[74] C. Albrecht, R. Hagenau, and A. Doring, "Cooperative software multithreading to enhance utilization of embedded processors for network applications", in *12th Euromicro Workshop on Parallel, Distributed and Network-Based Processing (PDP 2004)*, pp. 300–307, 2004.

[75] J. Nieh and M. S. Lam, "A SMART scheduler for multimedia applications", *ACM Trans. Comput. Syst.*, vol. 21, n. 2, pp. 117–163, 2003.

[76] A. Chandra, M. Adler, P. Goyal and P. Shenoy, "Surplus Fair Scheduling: A Proportional-Share CPU Scheduling Algorithm for Symmetric Multiprocessors", in *Symposium on Operating System Design and Implementation*, pp. 45–58, October 2000.

[77] P. Faraboschi, G. Brown, J. Fisher, G. Desoli and F. Homewood, "Lx: A Technology Platform for Customizable VLIW Embedded Processing", in *International Symposium on Computer architecture (ISCA)*, June 2000.

[78] J. Hill and D. Culler, "A wireless embedded sensor architecture for system-level optimization", Technical report, U.C. Berkeley, 2001.

[79] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker and P. Roussel, "The Microarchitecture of the Pentium 4 Processor", *Intel Technology Journal*, Q1 2001.

[80] R. Kessler, "The Alpha 21264 Microprocessor", *IEEE Micro*, vol. 19, n. 1, pp. 24–36, March/April 1999.

[81] D. Oehmke, N. Binkert, T. Mudge and S. Reinhardt, "How to Fake 1000 Registers", in *MICRO*, pp. 7–18, 2005.

[82] T. Baker, J. Snyder and D. Whalley, "Fast Context Switches: Compiler and Architectural Support for Preemptive Scheduling", in *Microprocessors and Microsystems*, pp. 35–42, September 1995.

[83] J. Redstone, S. Eggers, and H. Levy, "Mini-threads: Increasing TLP on small-scale SMT processors", in *HPCA '03: Proceedings of the The Ninth International Symposium on High-Performance Computer Architecture*, pp. 19–30, 2003.

[84] V. Barthelmann, "Inter-Task Register-Allocation for Static Operating Systems", in *LCTES-SCOPES*, June 2002.

[85] A. G. Dean, "Software Thread Integration and Synthesis for Real-Time Applications", in *Conference on Design, Automation and Test in Europe (DATE)*, pp. 68–69, 2005.

[86] A. G. Dean, *Software thread integration for hardware to software migration*, PhD thesis, 2000.

[87] M. Merten, A. Trick and R. Barnes, "An Architectural Framework for Runtime Optimization", *IEEE Transactions on Computers*, vol. 50, n. 6, pp. 567–589, 2001.

[88] T. Sherwood, E. Perelman, G. Hamerly S. Sair and B. Calder, "Discovering and exploiting program phases", *IEEE Micro*, vol. 23, n. 6, pp. 84–93, November-December 2003.

[89] M. Kandemir, J. Ramanujam, M.J. Irwin, N. Vijaykrishnan, I. Kadayif and A. Parikh, "A compiler-based approach for dynamically managing scratch-pad memories in embedded systems", *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, n. 2, pp. 243– 260, February 2004.

[90] M. J. Absar and F. Catthoor, "Compiler-Based Approach for Exploiting Scratch-Pad in Presence of Irregular Array Access", in *DATE '05: Proceedings of the conference on Design, Automation and Test in Europe*, pp. 1162–1167, 2005.

[91] P. Francesco, P. Marchal, D. Atienza, L. Benini, F. Catthoor and J. M. Mendias, "An integrated hardware/software approach for run-time scratchpad management", in *Design Automation Conference (DAC)*, pp. 238–243, 2004.

[92] M. Verma, L. Wehmeyer and P. Marwedel, "Dynamic overlay of scratchpad memory for energy minimization", in *CODES+ISSS*, pp. 104–109, September 2004.

[93] S. Udayakumaran and R. Barua, "Compiler-Decided Dynamic Memory Allocation for Scratch-Pad Based Embedded Systems", in *ACM International Conference on Compilers, Architecture, and Synthesis for Embedded Systems*, November 2003.

[94] R. Banakr, S. Steinke, B. Lee, M. Balakrishnan and P. Marwedel, "Scratchpad memory: design alternative for cache on-chip memory in embedded systems", in *CODES*, pp. 73–78, May 2002.

[95] R. Heckmann, M. Langenbach, S. Thesing and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools", *Proceedings of the IEEE*, vol. 91, n. 7, pp. 1038– 1054, July 2003.

[96] S. Baase and A.V. Gelder, *Computer Algorithms*, Addison-Wesley, Boston, MA, 2000.

[97] S. Martello and P. Toth, *Knapsack Problems: Algorithms and Computer Implementations*, John Wiley & Sons Inc, 1990.

[98] M. Chiodo, P. Giusto, A. Jurecska, H. Hsieh, A. Sangiovanni-Vincentelli and L. Lavagno, "Hardware-Software Codesign of Embedded Systems", *IEEE Micro*, vol. 14, n. 4, pp. 26–36, 1994.

[99] R. Kirner and P. Puschner, "Transformation of Path Information for WCET Analysis during Compilation", in *Euromicro Conference on Real-Time Systems (ECRTS)*, page 29, 2001.

[100] K. Flautner, N. Kim, S. Martin, D. Blaauw and T. Mudge, "Drowsy caches: simple techniques for reducing leakage power", in *International Symposium on Computer Architecture (ISCA)*, pp. 148–157, May 2002.

[101] J. S. Hu, A. Nadgir, N. Vijaykrishnan, M. J. Irwin and M. Kandemir, "Exploiting Program Hotspots and Code Sequentiality for Instruction Cache Leakage Management", in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 402–407, August 2003.

[102] P. Shivakumar and N. Jouppi, "CACTI 3.0: An Integrated Cache Timing, Power and Area Model", Technical report, Western Research Lab, 2001.

[103] C. Lee, M. Potkonjak and W. H. Mangione-Smith, "MediaBench: A Tool for evaluating and synthesizing multimedia and communications systems", in *International Symposium on Microarchitecture (MICRO)*, pp. 330–335, December 1997.

[104] M.R Guthaus, J. S. Ringenberg, D. Ernst, T.M. Austin, T. Mudge and R.B. Brown, "MiBench: A free, commercially representative embedded benchmark suite", in *WWC-4: Workshop on Workload Characterization*, pp. 3–14, December 2001.

[105] T. Austin, E. Larson and D. Ernst, "SimpleScalar: An infrastructure for computer system modeling", *IEEE Computer*, vol. 35, n. 2, pp. 59–67, February 2002.

[106] V. Stojanovic and V.G. Oklobdzija, "Comparative analysis of master-slave latches and flip-flops for high-performance and low-power systems", *IEEE Journal of Solid-State Circuits*, vol. 34, n. 4, pp. 536 – 548, April 1999.

[107] D. Woo, M. Ghosh, E. Ozer, S. Biles and H.-H. Lee, "Reducing Energy of Virtual Cache Synonym Lookup using Bloom Filters", in *International Conference on Compilers, Architectures and Synthesis for Embedded Systems (CASES)*, pp. 179–189, October 2006.

[108] M. Cekleov and M. Dubois, "Virtual-address caches. Part 1: problems and solutions in uniprocessors", *IEEE Micro*, vol. 17, n. 5, pp. 64–71, September 1997.

[109] D. Tarjan, S. Thoziyoor and N. Jouppi, "CACTI 4.0: An Integrated Cache Timing, Power and Area Model", Technical report, HP Laboratories Palo Alto, June 2006.

[110] M. Vratonjic, B. Zeydel and V. Oklobdzija, "Low- and Ultra Low-Power Arithmetic Units: Design and Comparison", in *International Conference on Computer Design (ICCD)*, pp. 249–252, 2005.

[111] Y. Tan and III V. J. Mooney, "WCRT analysis for a uniprocessor with a unified prioritized cache", in *Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES)*, pp. 175–182, 2005.

[112] D. C. Sastry and M. Demirci, "The QNX Operating System", *Computer*, vol. 28, n. 11, pp. 75–77, 1995.

[113] G. Reinman and N. Jouppi, "An Integrated Cache Timing and Power Model", Technical report, Western Research Lab, 1999.

[114] C. Nagendra, M.J. Irwin and R.M. Owens, "Area-time-power tradeoffs in parallel adders", *IEEE Transactions on Analog and Digital Signal Processing*, vol. 43, n. 10, pp. 689 – 702, October 1996.

[115] Alan Jay Smith and Rafael H. Saavedra, "Measuring Cache and TLB Performance and Their Effect on Benchmark Runtimes", *IEEE Trans. Comput.*, vol. 44, n. 10, pp. 1223–1235, 1995.

[116] N. Binkert, R. Dreslinski, L. Hsu, K. Lim, A. Saidi and S. Reinhardt, "The M5 Simulator: Modeling Networked Systems", *IEEE Micro*, vol. 26, n. 4, pp. 52–60, 2006.

[117] G. Byrd and M. Holliday, "Multithreaded processor architectures", *IEEE Spectrum*, August 1995.

[118] D. Bovet and M. Cesati, *Understanding the Linux Kernel (2nd Edition)*, OReilly, 2002.

[119] J. Fisher, P. Faraboschi and C. Young, *Embedded Computing: A VLIW Approach to Architecture, Compilers and Tools*, Morgan Kaufmann, 2005.

[120] A. Aho, R. Sethi and J. Ullman, *Compilers: Principles, Techniques and Tools*, Addison-Wesley, 1986.

[121] R. Mahajan, "Thermal management of CPUs: A perspective on trends, needs and opportunities", in *Keynote presentation, THERMINIC-8*, October 2002.

[122] L.T. Yeh and R. Chu, *Thermal management of microelectronic equipment*, American Society of Mechanical Engineers, 2001.

[123] S.Gunther, F. Binns, D.M. Carmean and J.C. Hall, "Managing the impact of increasing microprocessor power consumption", in *Intel Technology Journal*, 2001.

[124] Y.Zhang, "HotLeakage: A temperature-aware model of subthreshold and gate leakage for architects", Technical report, CS-2003-05, University of Virginia, 2003.

[125] J. Srinivasan and S. Adve, "Predictive dynamic thermal management for multimedia applications", in *ICS*, pp. 109–120, 2003.

[126] K. Skadron, M. Stan, W. Huang, S. Velusamy, K. Sankaranarayanan and D. Tarjan, "Temperature-aware computer systems: Opportunities and challenges", *IEEE Micro*, vol. 23, n. 6, pp. 52–61, Nov.-Dec. 2003.

[127] D. Brooks and M. Martonosi, "Dynamic Thermal Management for High-Performance Microprocessors", in *International Symposium on High-Performance Computer Architecture (HPCA)*, pp. 171–182, 2001.

[128] Huzefa Mehta, Robert Michael Owens, Mary Jane Irwin, Rita Chen and Debashree Ghosh, "Techniques for low energy software", in *International Symposium on Low Power Electronics and Design (ISLPED)*, pp. 72–75, 1997.

[129] W. Ye, N. Vijaykrishnan, M. Kandemir and M. J. Irwin, "The design and use of simplepower: a cycle-accurate energy estimation tool", in *Design Automation Conference (DAC)*, pp. 340–345, June 2000.

[130] M. Kandemir, N. Vijaykrishnan, M.J. Irwin, W. Ye and I. Demirkiran, "Register relabeling: A post compilation technique for energy reduction.", in *Workshop on Compilers and Operating Systems for Low Power (COLP)*, Philadelphie, PA, USA, October 2000.

[131] J.C. Ku, S. Ozdemir, G. Memik and Y. Ismail, "Thermal Management of On-Chip Caches Through Power Density Minimization", in *MICRO*, pp. 283–293, 2005.

[132] D. Atienza, P. Raghavan, J. Ayala, G. De Micheli, F. Catthoor, D. Verkest and M. Lopez-Vallejo, "Compiler-Driven Leakage Energy Reduction in Banked Register Files", *Lecture Notes in Computer Science*, vol. 4148, n. 1, pp. 107–116, September 2006.

[133] S.H.K. Narayanan, M. Kandemir and O. Ozturk, "Compiler-Directed Power Density Reduction in NoC-Based Multi-Core Designs", in *ISQED*, pp. 570–575, 2006.

[134] J. Donald and M. Martonosi, "Techniques for Multicore Thermal Management: Classification and New Exploration", in *International Symposium on Computer Architecture (ISCA)*, pp. 78–88, 2006.

[135] S. Heo, K. Barr and K.Asanovic, "Reducing power density through activity migration", in *ISLPED*, pp. 217–222, August 2003.

[136] K. Patel, W. Lee and M. Pedram, "Active bank switching for temperature control of the register file in a microprocessor", in *Great Lakes Symposium on VLSI Systems (GLSVLSI)*, pp. 231–234, 2007.

[137] Peng Li, Yangdong Deng and Lawrence T. Pileggi, "Temperature-Dependent Optimization of Cache Leakage Power Dissipation", in *International Conference on Computer Design (ICCD)*, pp. 7–12, 2005.

[138] R. Mukherjee and S. Memik, "Systematic temperature sensor allocation and placement for microprocessors", in *Design Automation Conference (DAC)*, pp. 542–547, 2006.

[139] E. Kursun, C. Cher, A. Buyuktosunoglu and P. Bose, "Investigating the Effects of Task Scheduling on Thermal Behavior", in *Workshop on Temperature-Aware Computer Systems (TACS)*, June 2006.

[140] L. Babel, H. Kellerer and V. Kotov, "The k-partitioning problem", *Mathematical Methods of Operations Research*, vol. 47, n. 1, pp. 59–82, Feb. 1998.

[141] E. G. Coffman, *Computer and Job-shop Scheduling Theory*, John Wiley & Sons Inc, 1976.