FLORIDA STATE UNIVERSITY

COLLEGE OF ARTS AND SCIENCES

A PRESENTATION AND LOW-LEVEL ENERGY USAGE ANALYSIS OF TWO LOW-POWER ARCHITECTURAL TECHNIQUES

By

PETER GAVIN

A Dissertation submitted to the
Department of Computer Science
in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Degree Awarded:
Spring Semester, 2015

Peter Gavin defended this dissertation on December 11, 2014.
The members of the supervisory committee were:

David Whalley
Professor Co-Directing Dissertation

Gary Tyson
Professor Co-Directing Dissertation

Linda DeBrunner
University Representative

Andy Wang
Committee Member

ACKNOWLEDGMENTS

Thanks to my advisors, David Whalley and Gary Tyson, first for piquing my interest in computer architecture, and then for guiding me through the process of completing my dissertation. Many other people, including Magnus Själander, Alen Bardizbanyan, Ian Finlayson, Brandon Davis, Ryan Baird, and Stephen Hines have helped me with various aspects of this dissertation. My mother, Marie France, has been tremendously supportive and reassuring through the most stressful of times. And last but certainly not least, my fiancee Liz has provided immeasurable love and support, and I would not have been able to complete this dissertation without her.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

In recent years, the need to reduce the power and energy requirements of computer microprocessors has increased dramatically. In the past, microprocessor architects succeeded in improving the performance of their designs by increasing clock frequency, and building wider and deeper pipelines. These design choices inevitably lead to increased power and energy usage, and thus increased heat dissipation. With the proliferation of battery-powered embedded and mobile computer systems, it is necessary to reduce energy usage without sacrificing performance.

This dissertation analyzes two architectural techniques that are designed to reduce the energy usage required to complete computational tasks, without impacting performance. The first technique is the Tagless-Hit Instruction Cache (TH-IC), which reduces energy used for fetching instructions by disabling several large fetch-related structures when it can be guaranteed they are not needed. The second is Static Pipelining, which reduces power by moving much of the pipeline control from pipeline logic to the compiler. These techniques have previously been studied with high level, architectural models based on the Sim-Wattch simulator. Such models estimate total energy usage by attributing a usage to various architectural events. The energy for each event must be derived from actual physical models of those components, and will be inaccurate if energy usage is dependent on factors that cannot be summarized by a single energy value per event. In addition, issues such as circuit timing and fabrication technology cannot be considered by the model without designing the real circuit it represents.

This dissertation presents an analysis of physical models of a traditionally-architected 5-stage OpenRISC processor, an implementation of the TH-IC, and a statically pipelined processor, all of which have been created from scratch using a hardware definition language and computer aided design tools. The OpenRISC processor serves as a baseline for comparison versus the statically pipelined processor. Additionally, the RISC processor is examined both with and without the TH-IC. Accurate estimates of energy usage and timing are derived using these models.

CHAPTER 1

INTRODUCTION & MOTIVATION

In recent years, the need to reduce the power and energy requirements in computer microprocessors has increased dramatically. In the past, microprocessor architects succeeded in improving the performance of their designs by increasing clock frequency, and building wider and deeper pipelines. These design choices inevitably lead to increased power and energy usage, and thus increased heat dissipation. Eventually, this heat dissipation can exceed the rate at which heat can feasibly be removed from the chip, a limit often referred to as the power-wall. With the proliferation of battery-powered embedded and mobile computer systems, it is necessary to reduce energy usage without sacrificing performance. Additionally, power represents a significant fraction of the operating costs of datacenters, and the cost of operating its cooling equipment is clearly related to the power dissipated by the computational equipment.

Unfortunately, estimating the power comsumption of complex dynamic systems such as ASIC microprocessors is difficult. A simple power measurement technique, used by Sim-Wattch, counts events as they happen in the various components of the processor. While simple to implement, such models assume that a given event dissipates the same energy each time it occurs, which may not be the case. In addition, when building such a model, it is difficult to account for every source of energy dissipation, and to estimate the energy used by each of the wide variety of structures found in a processor. Finally, separate event sources may cause activity in overlapping areas of the design, and thus the energy used when both events occur may not be the sum of the energies used when only one of the events occur.

Among the most accurate methods for estimating power usage is by simulating the circuit directly. This entails simulating the transistors and wires that compose the circuit. But since the circuit is digital, the voltage at any point in the circuit will almost always be one of two voltages, and will only intermittently be at a value between them. This means that power can be modelled easily by simply accumulating the energy needed to (dis)charge the capacitance of the wires each time the voltage is switched. However, creating such a model requires having a (nearly) complete design of the circuit, such as you would provide to an ASIC manufacturer for fabrication. For instance, using the Synopsys tool suite, a circuit specified in VHDL can then be synthesized to a netlist using the Design Compiler. While providing a very detailed picture of power and energy usage within the processor, these simulations are time consuming compared to higher level simulations like Sim-Wattch. However, as computational resources have grown, such simulations are becoming more feasible.

This dissertation presents a study of two techniques that reduce the power usage required to complete computational tasks, without impacting performance. The first technique is the Tagless-Hit Instruction Cache, which reduces energy used for fetching instructions by disabling several large structures when it can be guaranteed they are not needed. The second is Static Pipelining, which reduces power by moving much of the pipeline control from pipeline logic to the compiler. These techniques have been studied with high level models such as Sim-Wattch, but have not been analyzed at a physical level.

In order to implement the high-detail simulations of these techniques, a configurable, component-based framework for low-level architectural exploration called CARPE (Computer Architecture Research Platform for Exploration)[1] has been designed as part of this dissertation. This framework provides a number of commonly used processor components, such as caches, branch predictors, and so on, that have been designed to allow easy integration. The framework uses a text-based configuration system based on the Linux kernel `kconfig` system,[20] allowing easy modification of various parameters such as cache size and associativity, branch predictor type, and so on.

This dissertation presents a study these techniques by implementing and simulating them at the netlist level, and estimating the energy used for a number of benchmarks, using the CARPE framework.

CHAPTER 2

CARPE (COMPUTER ARCHITECTURE RESEARCH PLATFORM FOR EXPLORATION)

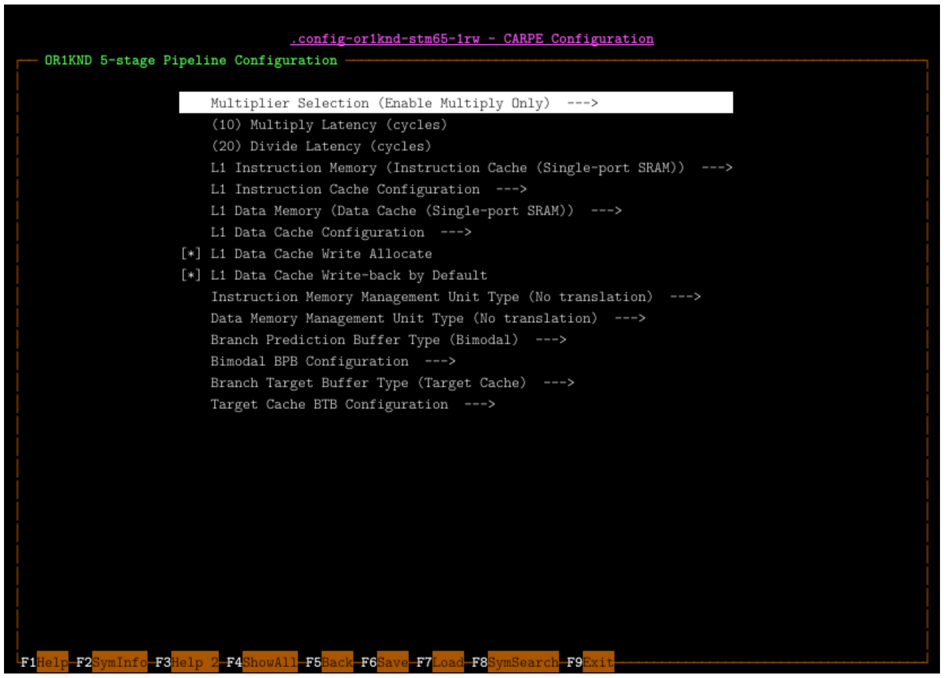
CARPE is a framework designed to provide easy configuration and elaboration of CPU design netlists, along with simulators for the netlists. The framework provides several components that can be reused between designs, including L1 instruction and data caches, branch prediction buffers (BTBs), branch target buffers (BTBs), and arithmetic units. This reuse allows for fair comparisons between designs, and quicker implementation of CPU designs, as implementing these subcomponents can be quite difficult and time consuming. CARPE also includes a system for collecting and exporting run-time statistics, and can export execution traces to an external program or script for processing in parallel with the simulator execution.

2.1 Configuration

The framework uses a `kconfig`-based configuration system, shown in Figure 2.1. This configuration system can be used to modify various parameters, such as cache and branch predictor implementations, pipeline features, testbench parameters, and simulation and synthesis tool configurations.

2.2 Components

The CARPE framework includes a traditionally architected five-stage pipeline capable of executing the integer subset of the OpenRISC instruction set architecture.[22] The block diagram of this processor is shown in Figure 2.2. The pipeline architecture is based on Hennessy and Patterson's DLX design, and includes instruction and data caches, branch prediction, support for



```
                     .config-or1knd-stm65-1rw - CARPE Configuration
 OR1KND 5-stage Pipeline Configuration

                Multiplier Selection (Enable Multiply Only)  --->
                (10) Multiply Latency (cycles)
                (20) Divide Latency (cycles)
                L1 Instruction Memory (Instruction Cache (Single-port SRAM))  --->
                L1 Instruction Cache Configuration  --->
                L1 Data Memory (Data Cache (Single-port SRAM))  --->
                L1 Data Cache Configuration  --->
            [*] L1 Data Cache Write Allocate
            [*] L1 Data Cache Write-back by Default
                Instruction Memory Management Unit Type (No translation)  --->
                Data Memory Management Unit Type (No translation)  --->
                Branch Prediction Buffer Type (Bimodal)  --->
                Bimodal BPB Configuration  --->
                Branch Target Buffer Type (Target Cache)  --->
                Target Cache BTB Configuration  --->

F1Help F2SymInfo F3Help 2 F4ShowAll F5Back F6Save F7Load F8SymSearch F9Exit
```

Figure 2.1: CARPE Configuration

integer multiplication and division, and exception handling. This design is used as a baseline for comparison in the studies described in later chapters of this dissertation.

CARPE includes reusable L1 instruction and data caches. The instruction cache supports line fill in critical-word-first order, and can bypass instructions from memory to the pipeline as they are being written to the cache. Sequential accesses that are within the same cache line can be satisfied without repeating the tag check, until the end of the line is reached.

The data cache also supports critical-word-first line fill, and can also bypass data from memory to the pipeline. The data cache uses a store buffer to allow back-to-back loads and stores without stalling, assuming no cache misses.

2.3 Simulation

In addition to providing component and processor implementations, CARPE also provides the ability to assemble the processor design from its components, synthesize the design, and create

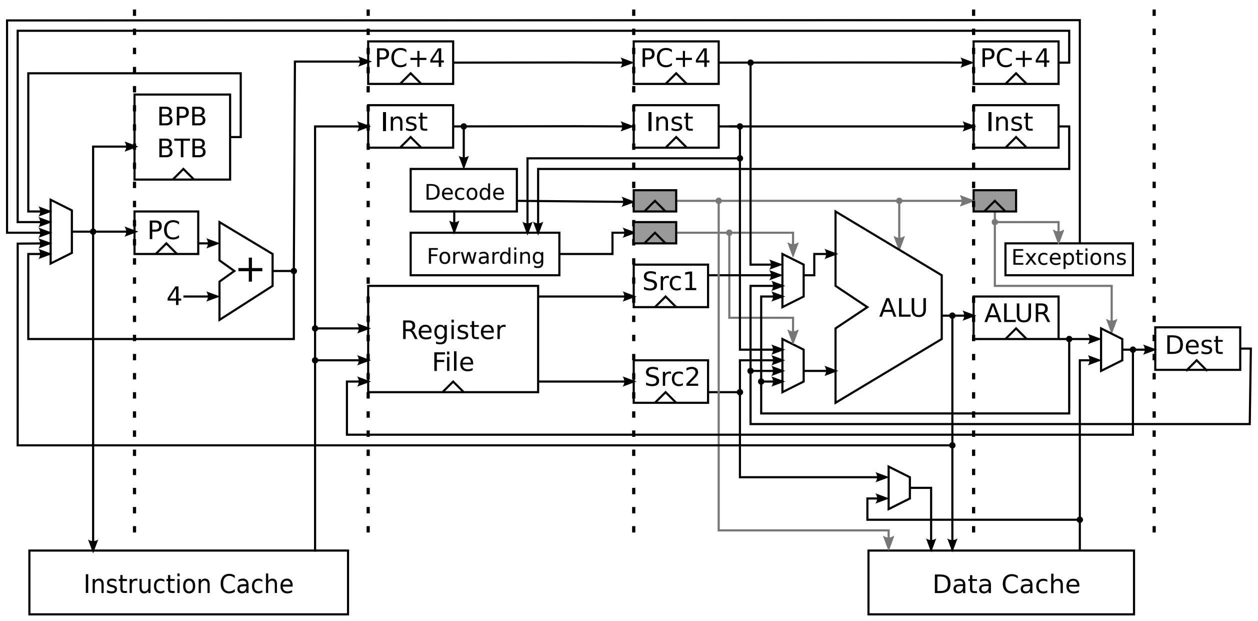


Figure 2.2: OpenRISC Pipeline Block Diagram

simulators for the design from both RTL and netlists, all from a single simple command line interface. The framework supports various VHDL simulation tools, including GHDL, Synopsys VCS, and Cadence NCSIM. The framework currently only supports Synopsys Design Compiler for synthesis.

CARPE provides the ability to gather run-time traces of events throughout the pipeline. Events can be traced using a *monitor*, which is an instantiable component that watches for changes to the value held by a particular signal. When a value change occurs, a record of the event, including an event identifier, a timestamp, and an associated value are output to a file or Unix fifo. This allows an external program to process the event trace to generate aggregated statistics of the simulation.

The framework generates a driver script for each simulator. The driver script starts the simulator using the correct parameters, and possibly a co-simulator that will be used to verify execution. If the cosimulator is available, the trace generated through the simulation monitors is compared against the trace of committed instructions produced by the cosimulator. This verification occurs in a process separate from both the main RTL/netlist simulator and the cosimulator, and thus can be done in parallel.

As previously mentioned, CARPE can simulate a design compiled either from RTL or from a synthesized netlist. Netlist simulations are opaque, and unlike RTL simulations, cannot be enhanced with monitors that trace internal events. Thus, when a netlist simulation is performed, the original RTL from which the netlist was synthesized is simultaneously simulated, and the external behavior of the netlist is compared to that of the RTL. This allows partial verification of the correctness of the netlist simulation, as well as allowing statistics to be generated from the RTL model during the simulation.

Since this framework is designed to perform a high-detail simulation of the processor, only the processor is simulated. Components that require a real system bus, such as hard disk drives, displays, keyboards, and so on are not supported. This choice was made in order to significantly reduce run time and the overall complexity of the simulator. However, it means that simulations must execute completely from main memory, without relying on the filesystem or console.

During the simulator initialization phase, the program to be executed is read from the disk file whose name is provided in the command line. The program is written to the simulated main memory, the processor's synchronous reset signal is asserted, and then the clock is started. The reset signal is then deasserted after one clock cycle, after which the processor starts executing instructions.

The benchmarks used in this study come from the Mibench benchmark suite. Only the integer instructions are supported by the pipeline, and thus the benchmarks that use floating-point cannot be simulated. Code necessary to initialize caches and set up the C runtime environment was added at the start of the benchmarks. This initialization code requires only a few thousand cycles to execute, and is negligible when compared to the millions of cycles required to complete the benchmark. Additionally, these benchmarks have been modified to remove use of system calls. This required that any input data used by the benchmark be statically included in the program binary itself. Since loading the program in memory occurs before the processor begins execution, loading the input data does not contribute to any of the statistics gathered by the simulator. However, the input data is typically only read once at the beginning of the program, and is on the order of kilobytes, and thus also contributes a negligible amount to the runtime of the benchmarks.

CHAPTER 3

TAGLESS-HIT INSTRUCTION CACHE

The TH-IC [14] was introduced as an effective alternative to an L0/filter cache for single-issue pipelines, with the goal of reducing instruction fetch energy. This chapter will provide background material on the TH-IC, as well as the preliminary work on improving its effectiveness, and extending its use to high-fetch-bandwidth pipelines.

3.1 Background

The primary motivator in targeting instruction fetch for energy reduction is that this stage of the pipeline requires accessing a number of large structures in order to maintain high throughput. A *level zero instruction cache* (L0-IC) reduces fetch energy when compared to using just a *level one instruction cache* (L1-IC), as an L1-IC is much larger, often containing a factor of at least 64 times as many instructions [17]. However, an L0-IC has much higher miss rates and each miss incurs a 1-cycle miss penalty as the L0-IC is checked before accessing the L1-IC. These miss penalties cause both a loss in performance and reduce some of the energy benefit from accessing the lower power L0-IC due to increased application execution time. The TH-IC holds instructions and provides them to the core in nearly the same fashion as an L0-IC, but has no associated miss penalty when the instruction is not present. Instead, it makes guarantees about when the next instruction to be fetched will be resident in the TH-IC by using a small amount of additional metadata. When such a guarantee cannot be made, the TH-IC is bypassed and the L1-IC is instead directly accessed. No tag check is performed when an instruction is guaranteed to reside within the TH-IC. In addition, I-TLB (Instruction Translation Lookaside Buffer) accesses are unnecessary on guaranteed hits as the page number of the virtual address is not used to index into the TH-IC. The energy savings for a TH-IC actually exceeds the savings from an L0-IC due to no execution time penalty on TH-IC

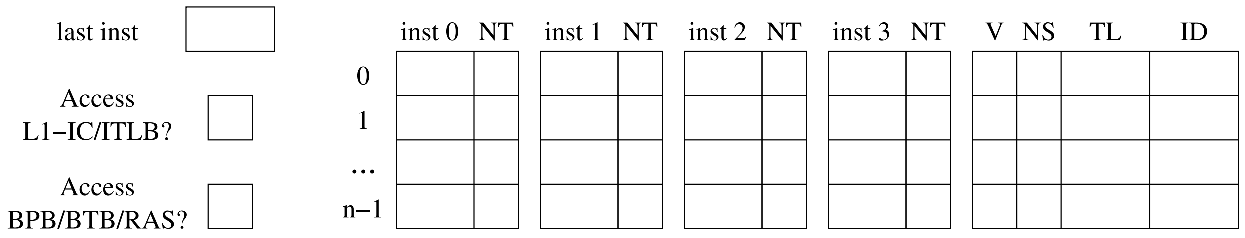


Figure 3.1: TH-IC Organization

misses and no tag checks or I-TLB accesses on TH-IC hits.

Figure 3.1 shows the organization of a TH-IC with a configuration of four instructions per cache line. There is a *next sequential* (NS) bit associated with each line that indicates that the next sequential line in memory is guaranteed to be the next sequential line in the TH-IC. Associated with each instruction is a *next target* (NT) bit indicating that the instruction is a direct ToC and that its target instruction resides within the TH-IC. These bits allow the processor to determine, while accessing a given instruction, whether the next instruction to be fetched will be a guaranteed TH-IC hit. Whenever an instruction is guaranteed to be present in the TH-IC, the bit labeled *"Access L1-IC/I-TLB?"* is cleared, and is set otherwise.

For instance, if the next instruction to be sequentially fetched follows the current instruction, and the current instruction is not the last in its line, then the next fetch is guaranteed to be present within the TH-IC, since the line will not be evicted before it is fetched. Otherwise, if the current instruction is the last in its line, we must check the line's NS bit. If the NS bit is set, then the next line is a guaranteed hit, and will be accessed from the TH-IC. If the NS bit is not set, then the next line is not a guaranteed hit, and it will be accessed directly from the L1-IC. At the end of the cycle in which the successive instruction is fetched, the L1-IC line containing it is written into the TH-IC, and the previous line's NS bit is set. As long as neither line in the pair is evicted, later traversals across the pair will result in guaranteed hits to the second line. If either line in the pair is evicted, the first line's NS bit will be cleared.

The NT bit plays a similar role in guaranteeing hits. However, it is used to determine whether or not the target of a direct transfer of control (ToC) is a guaranteed hit. For example, suppose the current instruction being fetched is a direct transfer of control, and the branch predictor determines that it should be taken. If the NT bit associated with the instruction is set, then the target is guaranteed to be present within the TH-IC. Otherwise, the target will be fetched from the L1-IC. At the end of the cycle in which the target is fetched, it will be written into the TH-IC, and the NT bit associated with the ToC will be set. If neither the ToC's line nor the target's line are evicted prior to the next time the ToC is fetched, then the NT bit will still be set. If the branch predictor again indicates that the ToC should be taken, the target will be accessed from the TH-IC.

In addition to the bits used to provide guarantees, the TH-IC tracks state to aid in invalidating these bits when they become obsolete. Each line has an associated TL field that indicates which lines should have its NT bits cleared when the corresponding line is evicted. Since the TH-IC is accessed in the same cycle as the L1-IC on potential misses to avoid invalidating metadata for false misses, the TH-IC can eliminate the high portion of the tag that is redundant to the L1-IC. These smaller tags are referred to as IDs. Despite containing additional types of metadata, the TH-IC metadata size per line is about the same size as the metadata in an L0-IC per line due to the significantly decreased tag size.

The following sections describe preliminary work on new extensions to the TH-IC that provide multiple instructions per cycle, as well as improving its energy usage reductions.

3.2 High Level Simulation

Before going into detail regarding microarchitectural innovations, we describe the processor configuration, benchmarks, simulation environment, and target instruction set used in this study. This information is referenced later in the paper when describing the design. As the currently predominant architecture in embedded systems is the ARM architecture, we have based our model on the ARM Cortex series of cores. The series includes a wide variety of cores, from simple, single-issue, in-order architectures with short pipelines, to complex, superscalar, out-of-order cores with pipelines of up to 20 stages. These cores are intended for low power usage scenarios, and can be implemented using synthesized logic and compiled RAMs, and without use of custom blocks. The new techniques presented in this paper can similarly be implemented without custom blocks, and using synthesized logic.

Our experimental configurations are shown in Table 3.1. For single issue experiments, we used a similar configuration to the one described in the TH-IC paper [14] to verify that the TH-IC results can be reproduced and to facilitate comparisons. We also used two more aggressive processor models: a two-way fetch, superscalar in-order model, and a four-way fetch, superscalar out-of-order model. As mentioned, all the models have been parameterized to closely represent the features found in the ARM Cortex series used in smartphones and tablets. We have attempted to use configurations similar to those described in the ARM online documentation [2].

The simulation environment used includes the SimpleScalar simulator [6, 3] with Wattch extensions [5]. Wattch assumes clock gating, and approximates the leakage in inactive processor components as using a constant percentage of their dynamic energy. The Wattch power model using CACTI [28] has been accepted for providing reasonably accurate estimates for simple structures. We have modified Wattch to allow this leakage to be configured, and run simulations assuming 10%, 25%, and 40% leakage. The benchmark suite used in this study is the MiBench suite [12]. We compiled all benchmarks using the GCC compiler included with the SimpleScalar toolset.

The structures we evaluate in this study include the TH-IC, L1-IC, I-TLB, BPB, and BTB. The TH-IC has a similar structure to that of a small, direct-mapped L0-IC, and its metadata can be stored in registers since the size of each metadata type is very small.

Similarly to the TH-IC study, we also used the MIPS/PISA instruction set. The formats used for direct unconditional ToCs are shown in Figures 3.2 and 3.3. The target address in the jump format is an instruction address. Each MIPS instruction is four bytes in size and aligned on a four byte boundary. In this paper we describe the *program counter* (PC) as referring to instruction addresses rather than byte addresses as the two least significant bits in a byte address are always zero.

We also modified the baseline instruction fetch architecture simulated by SimpleScalar in order to more realistically model a multiple issue processor. Our modified version of SimpleScalar has a configurable fetch width, allowing up to one, two, or four sequentially consecutive instructions to

Table 3.1: Experimental Configuration

Similar to	ARM Cortex-A5	ARM Cortex-A8	ARM Cortex-A15
Issue Style	In-order		Out-of-order
Fetch/Decode/Issue Width	1	2	4
Fetch Queue Size	1	3	7
Memory Latency	100 cycles		
Page Size	4 kB		
DTLB, I-TLB	10-entry, fully assoc.	32-entry, fully assoc.	32-entry, fully assoc.
L1-IC	4 kB, 2-way assoc	16 kB, 4-way assoc	32 kB, 2-way assoc
L1-DC	4 kB, 4-way assoc, 16 B line	16 kB, 4-way assoc, 64 B line	32 kB, 2-way assoc, 64 B line
L2-UC	128K, 8-way assoc	256K, 8-way assoc	512K, 16-way assoc
BPB	Bimodal, 256 entries	2-level, 512 entries	Comb., 1024 entries
BTB	256 entries	512 entries	1024 entries
Branch Penalty	2 cycles	4 cycles	8 cycles
RAS	8 entries		16 entries
Integer ALUs	1	2	4
Integer MUL/DIV	1	2	2
Memory Ports	1	2	3
FP ALUs	1	2	2
FP MUL/DIV	1	1	1
LSQ Size	2	4	8
RUU Size	2	4	8

6 bits	5 bits	5 bits	16 bits
opcode	rs	rt	branch offset

Figure 3.2: Conditional Direct Branch Format

6 bits	26 bits
opcode	target address

Figure 3.3: Unconditional Direct Jump Format

be fetched per cycle. We call the instructions fetched in a given cycle a *fetch group*. Each fetch group is always aligned to a boundary of its size. For instance, when simulating a four-wide fetch, each fetch group is aligned on a 16-byte boundary. If a transfer of control targets an instruction that is not at the start of a group, then the instructions in the group that reside prior to the target are not accessed. A beneficial side effect of this alignment is that exactly one L1-IC line is accessed per fetch cycle, which is not the case in the original SimpleScalar model. Once instructions are fetched, they are placed into a $(2 \times fetch\_width - 1)$-instruction deep queue. We assume that the number of instructions that can be issued from this queue per cycle is the same as the fetch width. We do not fetch instructions unless the queue has $fetch\_width$ free entries. In this way, the fetch architecture ensures that $fetch\_width$ instructions are available for issue each cycle, and yet does not fetch farther ahead than is immediately useful.

3.3 Designing a Multiple Fetch TH-IC

The original design of the TH-IC is limited in use to single issue microarchitectures. Such simple microarchitectures are useful in applications with demands for low power, but are less applicable when higher performance is demanded. Since multiple-issue processors are now becoming commonplace in low-power applications, we have designed a multiple fetch TH-IC that can be used in these more complex microarchitectures.

The layout and operation of the TH-IC is similar under multiple fetch pipelines as under single fetch. The output port on the TH-IC is widened to allow $fetch\_width$ adjacent instructions to be accessed each cycle. In addition, rather than tracking the position of the instruction fetched in the previous cycle, the position of the last issued instruction in the last fetch *group* is tracked. For instance, if on the next cycle the processor will fetch the group that sequentially follows the current fetch group, the position of the final instruction in the current group is stored. If the processor will fetch the target of a taken direct transfer of control, then the position of the transfer of control is stored, and *not* the position of the final instruction in its group. This position tracking behavior for taken direct transfers enables the correct NT bit to be set once the target is fetched.

A consequence of increasing the fetch width is that the number of opportunities for sequential intra-line fetches are less frequent. Suppose, for instance, that a basic block of code is being executed for the first time, and that it is several TH-IC lines in length. If the TH-IC line size is four instructions, a single issue processor will be able to take advantage of guaranteed hits for three out of four instructions in each line. However, when the fetch width is widened to four instructions, no hits can be guaranteed, as each sequential fetch will consume the entire line, and successive fetches will be from new lines. Thus it is quite beneficial to increase the instruction cache line size when using the TH-IC on multiple fetch processors.

3.4 Further Reducing Fetch Energy

While the instruction cache may be the dominating contributor to fetch energy dissipation, there are other structures that use significant additional power. These include the I-TLB and the structures involved in branch prediction.

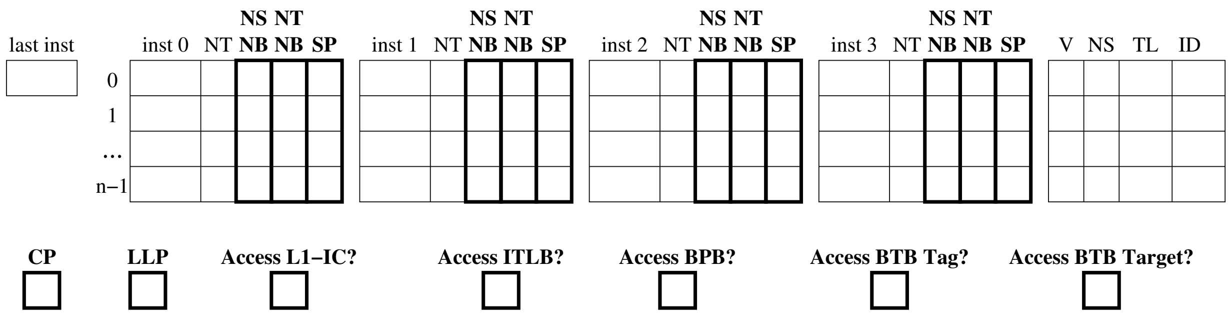
We have devised a new organization, as show in Figure 3.4, that stores metadata that is used to avoid accesses to the I-TLB and the branch related structures for multi-fetch architectures. Figure 3.5 illustrates the entire fetch engine once these changes have been put into place. The primary changes are to the TH-IC itself, and the introduction of a small amount of logic to detect small direct jumps, whose purpose and function will be explained later. Some additional control logic must be routed from the TH-IC to other structures, allowing the TH-IC to control when these structures are accessed. In a standard pipeline, the BTB, BPB, RAS, L1-IC, and I-TLB are accessed together in parallel, during a single pipeline stage. The TH-IC is also accessed in parallel with these structures, during the same pipeline stage. Data that is read from the L1-IC must be fed back to the TH-IC so it can be written at the end of the cycle. As the TH-IC may be accessed for a read and a write in the same cycle, it must be implemented using dual-ported memories.

While the TH-IC already disables the I-TLB much of the time, it provides opportunities to disable it even when the fetch will not occur from the TH-IC. This is facilitated by adding the CP, LLP, and SP bits as shown in Figure 3.4. There is one CP (*current page*) bit, which indicates current physical page is the same as the last one accessed from the I-TLB. This bit is necessary because, if the I-TLB is disabled, there may be occasions where the physical page number is not known, but fetching can proceed anyway. There is one LLP (*last line in page*) bit, which indicates that the last line in the TH-IC is also the last line in the page. This bit is used to detect when a sequential fetch leaves the current page. Finally, an SP (*same page*) bit is associated with each instruction, and indicates that the instruction is a direct transfer of control whose target is in the same page.

For disabling the branch prediction hardware, no new metadata is needed beyond that described in [13]. However, some of this metadata has been rearranged or is used in a different fashion. These changes are described in more detail in the following subsections.

3.4.1 Disabling I-TLB Accesses on TH-IC Misses

Disabling I-TLB accesses needs to be addressed within the context of the TH-IC, which disables the I-TLB on guaranteed hits. Consider Figure 3.6, which depicts a code segment that straddles a page boundary. There are three different ways that the application can switch between these two pages. First, the unconditional jump (instruction 3) in block 1 can transfer control to block 3 (instruction 7). Second, the conditional branch (instruction 9) in block 3 can transfer control to block 2 (instruction 4). Finally, there can be a sequential fetch within block 2 (instruction 5 to instruction 6). There is an NT bit associated with each instruction indicating if the associated instruction is a direct ToC and that its target instruction is resident within the TH-IC. There is



last inst	– TH–IC index of instruction last fetched
NT	– Next Target of current direct ToC is the TH–IC
NSNB	– Next Sequential inst is Not a conditional Branch
SP	– target of direct ToC is in the Same Page
NS	– Next Sequential line is in the TH–IC
NTNB	– Next Target is assumed to Not be a Branch
TL	– lines which have ToCs Transferring to this Line
V	– Valid bit
ID	– IDentifies L1–IC line loaded into TH–IC line
CP	– Current Page number from ITLB is still valid
LLP	– last line in TH–IC is the Last Line in the Page

Figure 3.4: New Metadata Organization

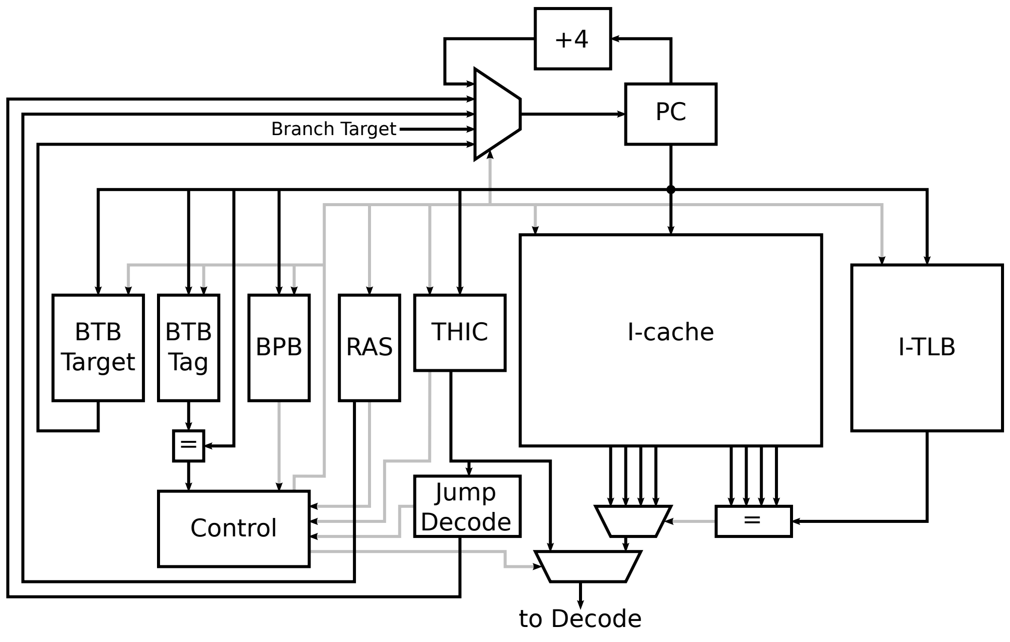


Figure 3.5: New Fetch Engine Block Diagram. Grey lines indicate control signals

an NS bit associated with each line of instructions indicating if the next sequential line is resident within the TH-IC. The figure shows the status of these bits after each of the three blocks has initially been executed. The I-TLB will not be accessed for the transitions 3→7, 9→4, and 5→6 when the NT bits associated with the first two transitions and the NS bit associated with the third transition are set. Note that when the next instruction is guaranteed to be in the TH-IC, the physical page number is not needed as there will be no tag check for guaranteed hits. Thus, as long as the loop continues to iterate, there will be no I-TLB accesses even though the page boundary is repeatedly being crossed.

We attempt to disable the I-TLB on TH-IC misses if the next instruction is guaranteed to remain in the same page. However, the processor needs to detect when a page boundary has been crossed on a TH-IC hit so that the next TH-IC miss will always access the I-TLB to obtain the current physical page number (PPN). We add an internal register to store the PPN immediately after it has been read from the I-TLB, and a *current page* bit (CP, shown in Figure 3.4) that indicates whether or not the stored PPN corresponds to the page of the instruction to be fetched. We additionally keep a single bit of state, called *"Access I-TLB?"*, that determines if the I-TLB should be enabled. We use this information to avoid I-TLB accesses when accessing the L1-IC. If the instruction resides on the same page, the most recently retrieved PPN is reused.

The CP bit is set each time the I-TLB is accessed as the PPN register is updated. Whenever a transition between instructions is made that crosses a page boundary and the next instruction is a guaranteed hit, then the CP bit will be cleared as the I-TLB is not accessed on TH-IC hits. The first instruction accessed that is a TH-IC miss when the CP bit is clear will result in an I-TLB access and the CP will be reset.

We store additional metadata to make guarantees when the next access is a TH-IC miss, but will reside in the same page so we can avoid additional I-TLB accesses. The analysis of this problem can be divided into distinct cases: (1) sequential accesses, (2) direct ToCs, (3) returns, and (4) other indirect ToCs. We attempt to disable I-TLB accesses in the first three cases. The following subsections describe when we can guarantee that the next instruction to be fetched for TH-IC misses will remain on the same page. I-TLB accesses can only be disabled in these cases when the

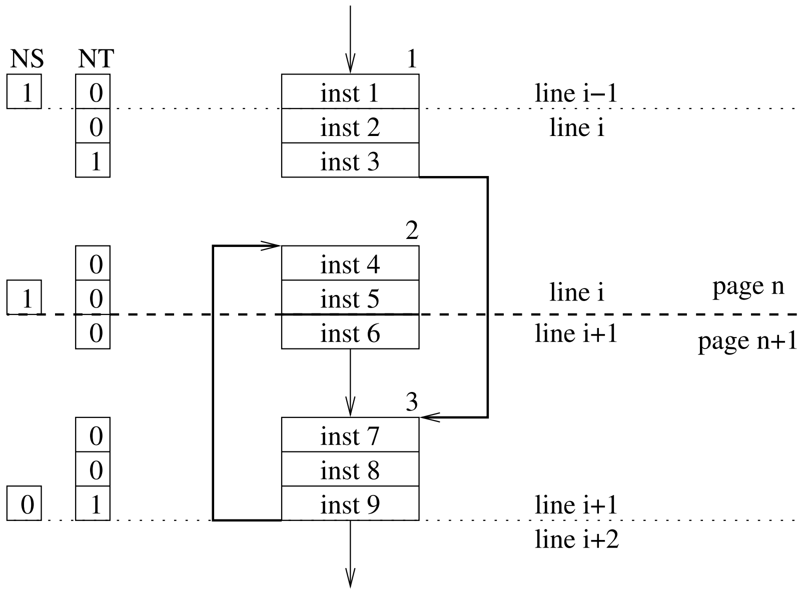


Figure 3.6: Example Loop Straddling a Page Boundary

CP bit is set.

3.4.2 Disabling the I-TLB for Sequential Accesses

If the current instruction is not the last instruction in the page, then the next sequential instruction accessed is guaranteed to reside on the same page. We perform this check in two steps. First, when the last line in the direct mapped TH-IC is replaced, a check is made to determine if the new line is the last line in the page, which is indicated using the *last line in page* bit (LLP, shown in Figure 3.4). Note that if the last line of the TH-IC is not being replaced, then the LLP bit is unchanged. Second, when there is a sequential fetch, the current instruction is checked to see if it is the last instruction in the TH-IC, which is accomplished by detecting if the portion of the virtual address containing the TH-IC index and the most significant bits of the line offset identifying the instruction within the line are all 1's. This check examines six bits for a TH-IC configured with sixteen 16-byte lines, as there are 64 instructions within the TH-IC. This second check only needs to be performed when the LLP bit is set. Figure 3.7 depicts the *line in page* and *inst in TH-IC* bits that must be accessed for these two checks. We have found that in 94.7% of all sequential fetches,

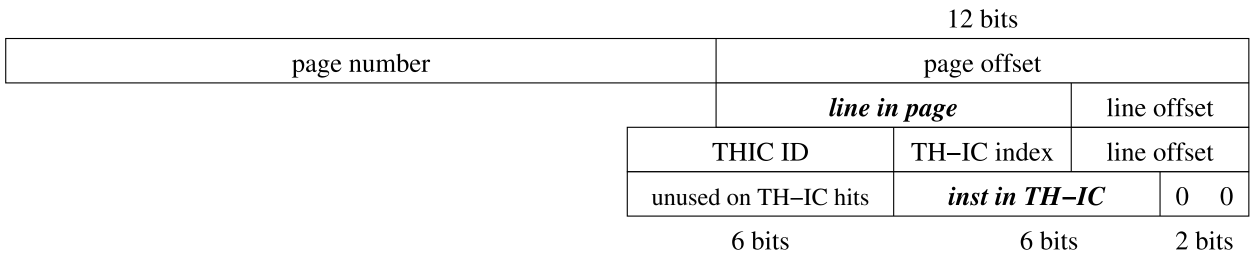


Figure 3.7: Address Fields to Check

the last line in the TH-IC is also not the last line in the page. This large percentage is due in part to the page size containing 16 times the number of instructions as the TH-IC in our configuration.

3.4.3 Disabling the I-TLB for Targets of Direct ToCs

If the target of a ToC is within the same page as the ToC instruction itself, then the I-TLB need not be accessed, as the current PPN register value can instead be used. Figure 3.8 shows how often each type of ToC occurred in the MiBench benchmark suite. Direct ToCs for the MIPS ISA include conditional branches, direct unconditional jumps, and direct calls, which comprised 14.2% of the total instructions and 93.3% of the ToCs. Figure 3.9 shows the ratios of how often these direct ToCs have targets to the same page, which occurs 93.7% of the time on average. We believe this frequency is high enough to warrant metadata to avoid I-TLB accesses when fetching these direct targets. Thus, we include a *same page* (SP) bit for each TH-IC instruction, depicted in Figure 3.4, that indicates when the instruction is a direct ToC and its target is on the same page. We check if the virtual page number returned by the I-TLB on the cycle when the target instruction is fetched is the same as the virtual page number used in the previous cycle for the ToC. If so, then the SP bit associated with the direct ToC instruction is set and subsequent fetches of the target fetch group when the ToC is taken need not access the I-TLB.

The processor does not always have to wait until the following cycle to set the SP bit when the direct ToC instruction is fetched from the TH-IC as a guaranteed hit. Instead, we can often exploit the faster TH-IC access time to determine within the same cycle if the target address is within the same page. Our experimental configuration used a four-kilobyte page size, which is a

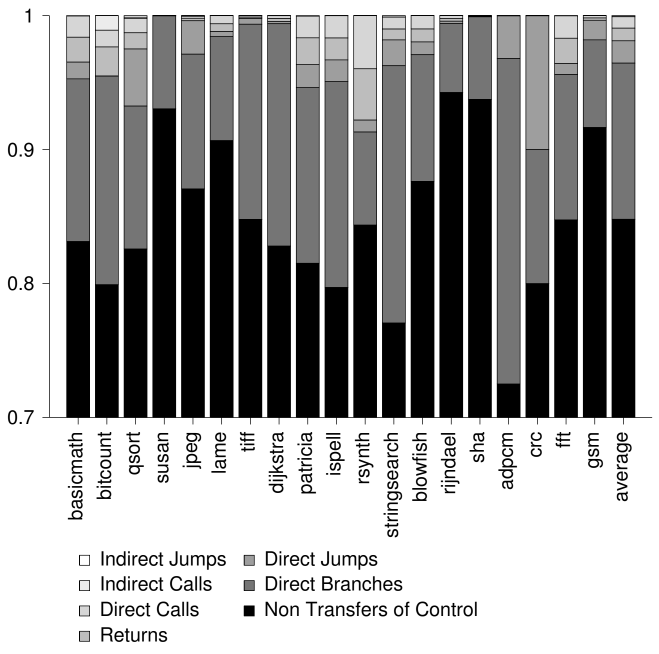


Figure 3.8: Distribution of Dynamic Instruction Types

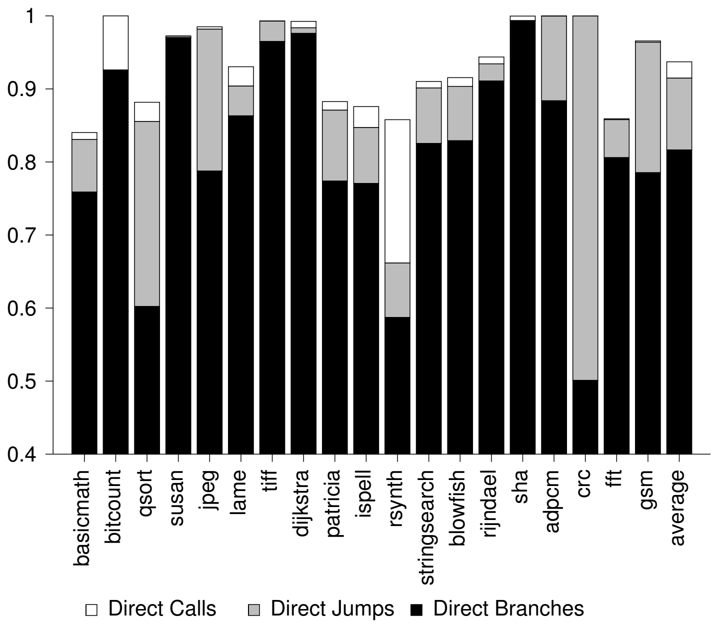


Figure 3.9: Direct ToCs with Same Page Targets

common size for many processors. A four-kilobyte page contains 1024 MIPS instructions, where each instruction is four bytes in size and aligned on a four byte boundary. Thus, only the ten most significant bits of the 12-bit page offset can have nonzero values for instruction addresses. If the instruction is a direct unconditional jump or call, and the 16 most significant target address bits shown in Figure 3.3 are the same as the corresponding bits of the *program counter* (PC), then the SP bit is set and the I-TLB is disabled for the next instruction.

3.4.4 Disabling the I-TLB for Targets of Returns

A *return address stack* (RAS) is used in our configuration to avoid mispredictions associated with return instructions. The processor can be made to compare the virtual page number of the target address in the top RAS entry with the virtual page number in the PC. This is possible because the top RAS entry is available very early in the cycle, and so the comparison can be performed without creating additional timing pressure. If they match and a return instruction is fetched on a TH-IC hit, then the I-TLB access on the next cycle is disabled.

3.4.5 Disabling BPB/BTB Accesses

On single-issue pipelines without delayed transfers of control, and on multiple-issue pipelines, branch prediction is a necessary feature in order to mitigate branch penalties. Branch prediction structures can be very large, and it is beneficial to avoid accessing them whenever possible. The BTB is particularly expensive to access, as each entry holds an address, and has an associated tag that must be compared with the ToC PC. We first describe techniques for disabling these structures on single fetch pipelines, then explain how they can be applied to multiple fetch pipelines.

3.4.6 Disabling BPB/BTB Accesses for Non-ToCs and Direct Jumps

The primary structures dealing with ToC speculation include the branch prediction buffer (BPB) and the branch target buffer (BTB). We store two sets of bits in the TH-IC that we use to disable these structures when they are not needed. These bits are the NSNB bits (next sequential not a branch) and NTNB bits (next target not a ToC). Each instruction has an NSNB bit and an

NTNB bit associated with it, as shown in Figure 3.4. Whenever the NSNB bit for an instruction is set, it indicates that the instruction that sequentially follows it is not a direct conditional branch. Similarly, when the NTNB bit for an instruction is set, it indicates that the instruction is a direct ToC, and that its target is *not* a direct conditional branch.[1]

We use these bits in the TH-IC by taking advantage of a single *predecode* bit that we have added to the L1-IC. Predecoding is used in many pipeline designs, such as to facilitate detection of calls and returns for activating the return address stack (RAS).

The bit we predecode prior to insertion in the L1-IC is the NB bit, which is set when the instruction is a not direct conditional branch. We fill the NSNB bits in the TH-IC from the predecoded NB bits in the L1-IC, as shown in Figure 3.10. The predecoded NB bits in the L1-IC are shifted over to initialize previous instructions' NSNB bits. We can also initialize NSNB bits in the previous TH-IC line, provided that the current line was reached by a sequential fetch, and not via a ToC.

We can avoid accessing the BPB and BTB on the following fetch if it is a guaranteed TH-IC hit, and one of the following is true. (1) The following fetch occurs sequentially after the current fetch, and the current instruction's NSNB bit is set. (2) The following fetch is the target of a direct ToC and the current instruction's NTNB bit is set.

Note that the BPB and BTB may now be disabled for direct unconditional jumps and calls. Due to the faster TH-IC access time, and the simple jump/call encoding used in the MIPS ISA, on TH-IC hits we have sufficient time to detect direct jumps and calls, and simply multiplex in the jump target before the next fetch, making the BPB and BTB irrelevant.

We also avoid updating the branch predictor for ToC instructions that do not activate the BPB and BTB. This can result in reduced contention in these structures, potentially improving branch prediction behavior.

The cost of this approach is extra predecode logic (checking for branch opcodes), the NB bits

[1]We used a simple branch predictor in this study that always mispredicts indirect jumps that are not returns. If an indirect branch predictor is used that stores target addresses for indirect jumps in the BTB, then the NSNB/NTNB bits could instead mean that the next fetched instruction is not a conditional branch and not an indirect jump other than a return.

added per L1-IC instruction, and the NSNB and NTNB bits per TH-IC instruction.

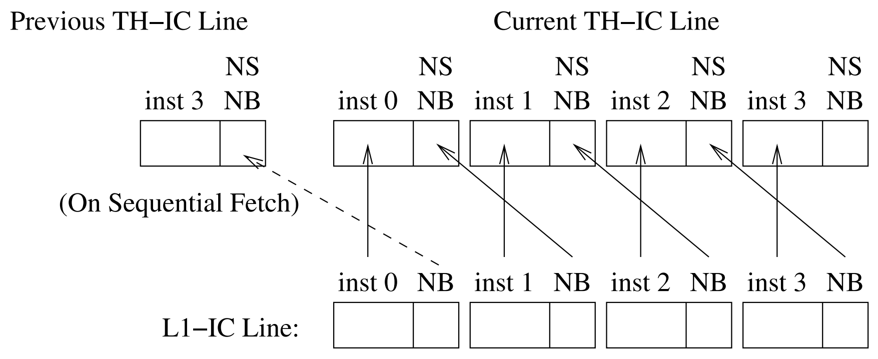


Figure 3.10: Filling NSNB Bits from L1-IC Predecode Bits

3.4.7 Disabling BTB Tag Array Accesses

The BTB is a simple cache of ToC targets, and like most caches, has a tag array that comprises a significant portion of its storage. However, since BTBs typically hold a single ToC target per entry, this tag array can be larger than data or instruction caches with similarly sized data arrays. Provided that the BTB has at least as many entries as the TH-IC has instructions, we can take advantage of a set NT bit to disable access to the BTB tag array on sequential fetches. If the fetch on the next cycle will sequentially follow the current instruction, and is either within the same line or the current line's NS bit is set, then we can examine the next sequential instruction's NT bit. If that bit is set, then we know that the BTB entry for that instruction cannot have been replaced since it was last accessed. In other words, if the BTB entry for the ToC is replaced, then the instruction itself within the TH-IC would have been replaced and the associated NT bit would have been cleared. As a result, a BTB hit is guaranteed when the NT bit is set, and we can avoid the BTB tag check that is normally required.

3.4.8 Disabling BPB/BTB Accesses with Multiple Fetch

When multiple instructions are fetched per cycle, we must slightly adjust the interpretation of the NSNB and NTNB bits. First, we associate the NSNB bits with each fetch group, rather than with each instruction. The total number of NSNB bits remains the same, at one bit per TH-IC instruction. For example, when the pipeline fetches four instructions per cycle, it will access four NSNB bits along with those instructions. However, each bit in this group will indicate the non-branch status of a corresponding instruction in the next sequential fetch group. Thus, when initializing these bits from the NB bits held in the L1-IC, we must shift them over by a whole fetch group, and not just a single instruction.

We also adjust the interpretation of NTNB bits. We still associate each NTNB bit with an instruction, but interpret each NTNB bit to mean that no instruction in the target fetch group (except those prior to the target instruction) is also a ToC. We could associate multiple bits per instruction, allowing individual instructions within the target fetch group to be disabled, but this is unlikely to provide much benefit. On average, branches will target the middle of a fetch group, so half these bits will not be used.

If a direct unconditional jump or call instruction is fetched from the TH-IC as a guaranteed hit, then the processor has time in the same cycle due to the faster TH-IC access time to extract the target address from the jump format, which is shown in Figure 3.3. Accessing the BTB in this case is unnecessary as the target address is already available, regardless of whether or not the target is on the same page. The BPB can also be disabled in this case as direct unconditional jumps and calls are always taken. Thus, when a sequentially accessed instruction, direct call or jump is a TH-IC hit and the NSNB bit for the previous instruction is set (meaning the current instruction is not a conditional branch), then both the BTB and BPB are disabled.

This strategy has the added effect that it is not necessary to insert direct jumps and calls into the BPB or BTB. Fewer accesses result in reduced contention in those tables, allowing more room for branches, and thus improving the branch prediction rate.

3.4.9 Variable Length Instruction Set Architectures

One of the dominant ISAs in current use is the Intel x86 ISA. Unlike the ISA used in this study, and unlike nearly every other ISA in common use, this ISA uses a variable-length instruction. While such an ISA may appear to make the TH-IC unusable for reducing fetch energy, this is not the case.

In order to adapt the TH-IC to variable-length ISAs, the key concept is to treat each *byte* of an x86 instruction as an individual, complete instruction. The fetch engine can then treat the first $N-1$ bytes of an N byte instruction as noops, and attributes the full effect of the complete instruction to the final byte. Metadata bits are then associated with each byte in the TH-IC, instead of each instruction. For instance, when setting the NT bit for a multi-byte branch instruction, the NT bit associated with the final byte of the branch will be set. With this organization, the multiple-fetch architecture described in this paper can be used to fetch multiple bytes per cycle, thus maintaining performance.

We did not study a TH-IC as just described with a variable-length ISA for this paper. The power savings would likely be marginally lower, as the amount of metadata storage required would be greater than for an ISA with a fixed, 32-bit instruction size.

3.5 High-level Simulation Results

The baseline shown in the graphs in this section assumes that the fetch associated structures (I-TLB, BPB, and BTB) are accessed every cycle. During our simulations we performed numerous checks to verify the validity of all guarantees. Our model accounts for the additional energy required to transfer an entire line from the L1-IC to the TH-IC, as opposed to just the instructions needed when no TH-IC is used. Our model also properly attributes the additional energy required for configurations that extend the L1-IC with predecoded data. Our experiments include configurations with L1-IC lines of different lengths, while keeping the L1-IC size and associativity constant. TH-IC lines are necessarily always the same length as L1-IC lines, but we included configurations with varying numbers of TH-IC lines.

As a share of total processor power, leakage power has increased with the downward scaling of fabrication technologies. As previously mentioned, in order to cover the range of potential process technologies, we have simulated three different leakage rates, 10%, 25%, and 40%.

Table 3.2 shows the experimentally determined lowest-energy configurations for each fetch width

Table 3.2: Optimal Configurations

Fetch Width	10% leakage	25% leakage	40% leakage
1	32/T8	16/T8	64/none
2	32/T8	32/T8	32/T8
4	32/T16	32/T16	32/T8

and leakage rate. In this table and the following figures, the notation "X/TY" refers to a configuration with an X-byte IC line size, and a TH-IC with Y lines. The notation "X/none" refers to a configuration with an X-byte IC line size, and *no* TH-IC.

Figures 3.11-3.19 show the total fetch energy for each pipeline, normalized to the best-case L1-IC-only configuration for each pipeline. The most obvious result shown by these graphs is that a larger TH-IC size results in more efficient usage of the other structures, but also increases TH-IC energy usage. The TH-IC hit rate generally increases with size, but eventually its power will exceed the power saved by disabling the other structures.

As expected, with higher leakage rates, the effectiveness of the TH-IC diminishes, and smaller TH-ICs are more effective than larger ones. In spite of this, the TH-IC provides benefit in all configurations, although that benefit may be miniscule at higher leakage rates. In addition, it is anticipated that technologies such as *high-k metal gate* (HKMG) will drastically reduce leakage power even as technology scales smaller. For instance, Global Foundries' 28nm HKMG Low-Power/High-Performance Platform (LPH), intended for use in high-end smartphones and tablets, reduces active power by up to 40%, and leakage currents by up to 90% when compared to earlier technologies [11]. Where power is a concern of much higher importance than performance, a very low leakage, low power technology will likely be used in combination with a low performance, single issue design, a situation in which the TH-IC is highly effective.

The following paragraphs describe the reduction in access rates of the individual components.

Figure 3.20 shows the resulting L1-IC accesses. This reduction in accesses is the primary contributor of energy savings, as the L1-IC is the largest of the fetch associated structures. We are able to avoid accesses to the L1-IC for between 77% and 88% of instructions in all configurations.

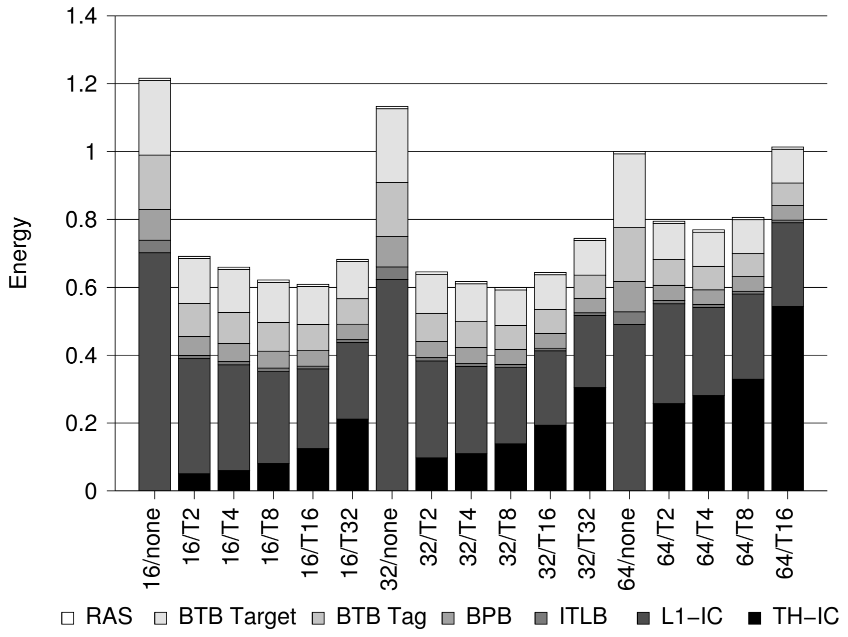


Figure 3.11: Fetch Energy, 1-wide Fetch, 10% Leakage

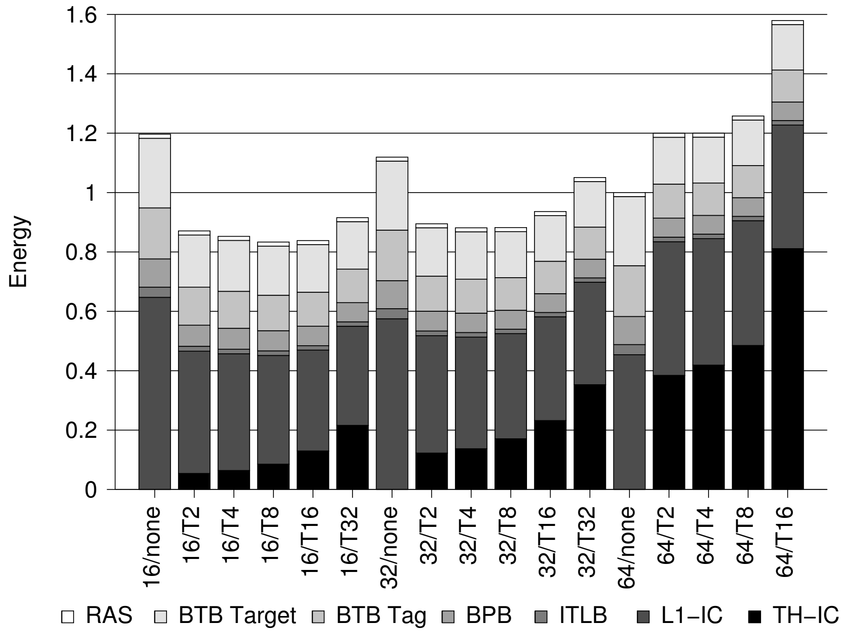


Figure 3.12: Fetch Energy, 1-wide Fetch, 25% Leakage

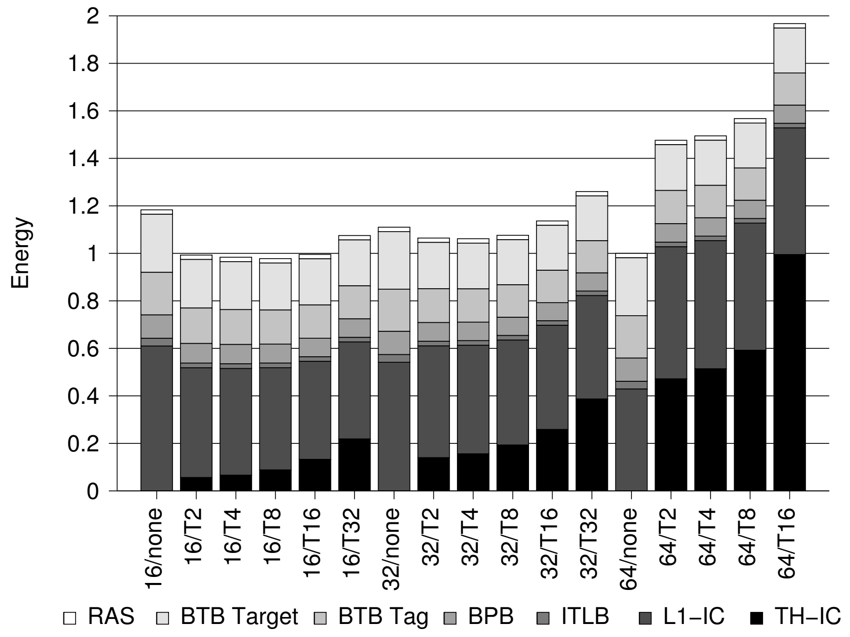


Figure 3.13: Fetch Energy, 1-wide Fetch, 40% Leakage

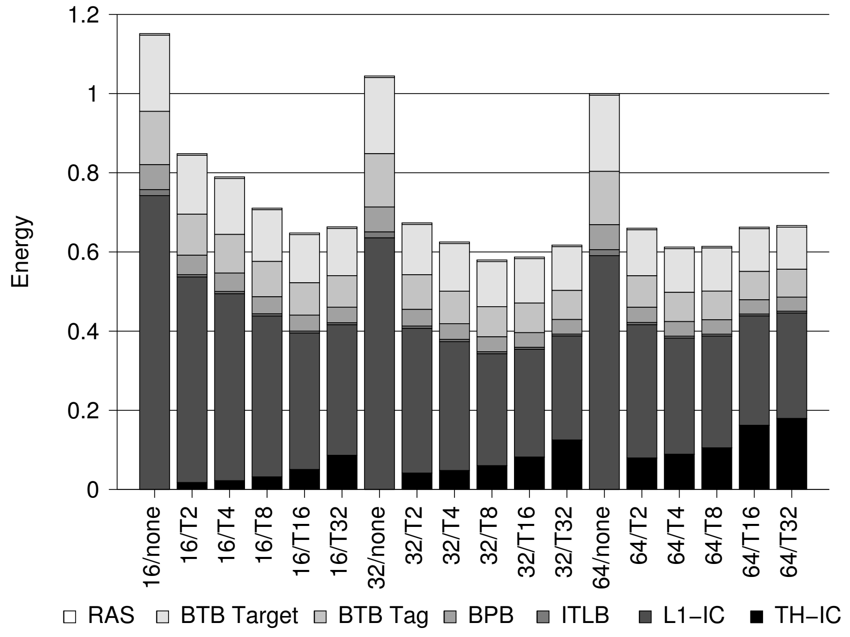


Figure 3.14: Fetch Energy, 2-wide Fetch, 10% Leakage

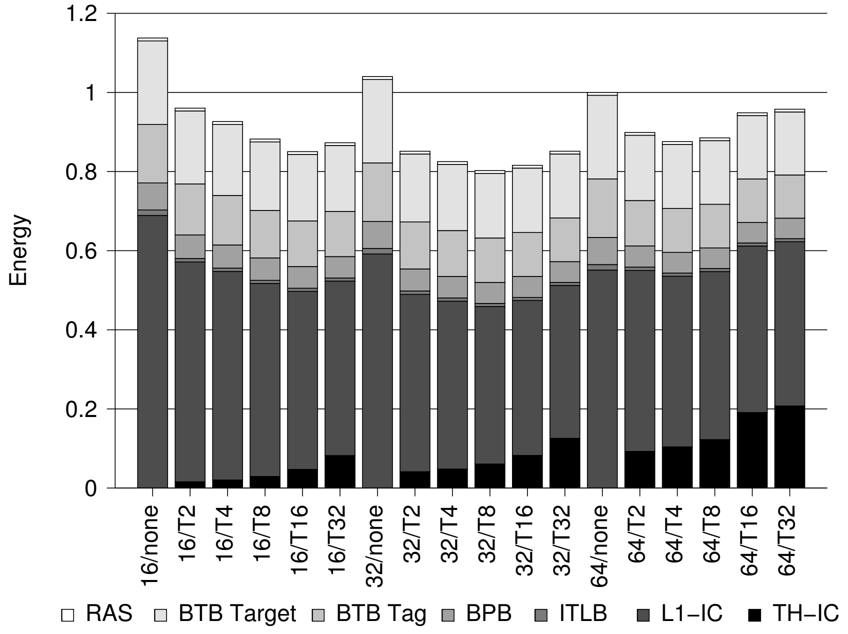


Figure 3.15: Fetch Energy, 2-wide Fetch, 25% Leakage

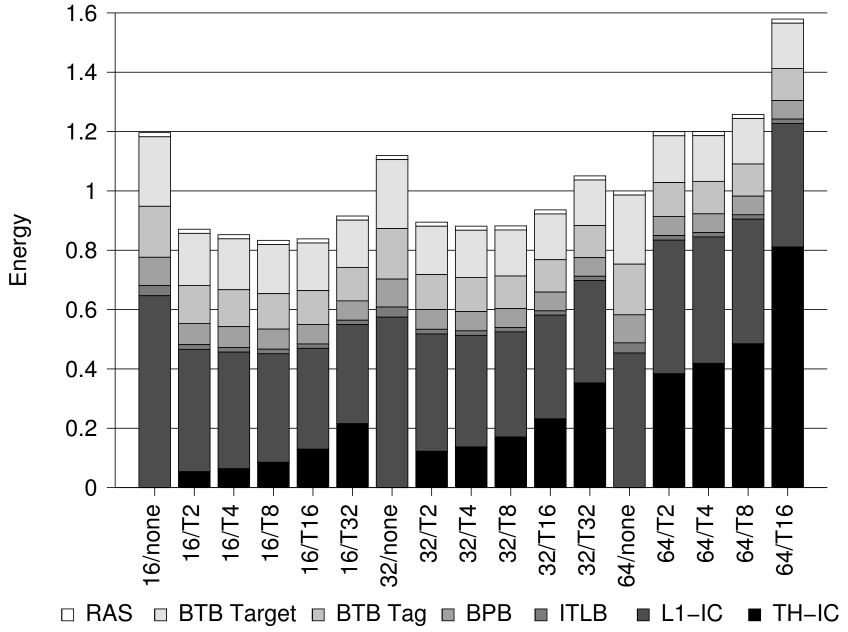


Figure 3.16: Fetch Energy, 2-wide Fetch, 40% Leakage

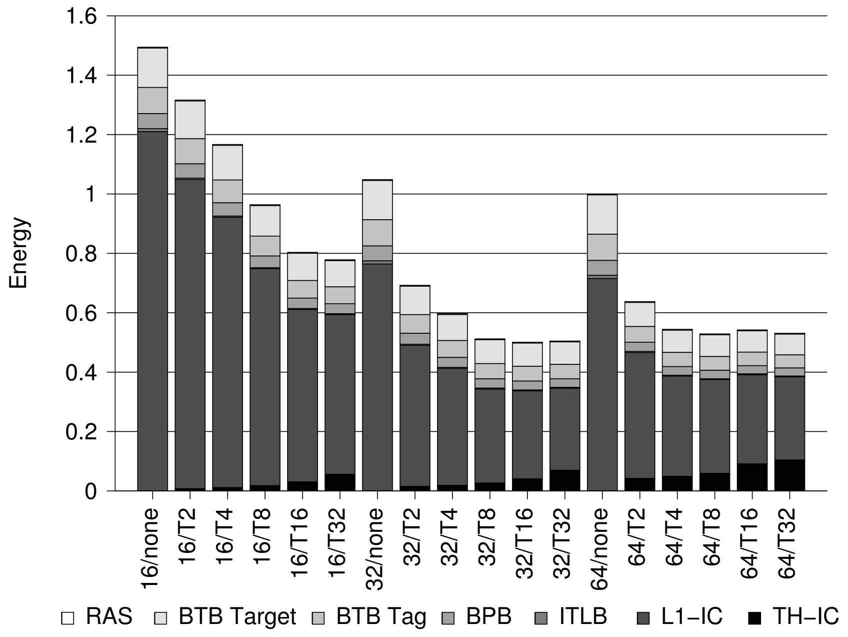


Figure 3.17: Fetch Energy, 4-wide Fetch, 10% Leakage

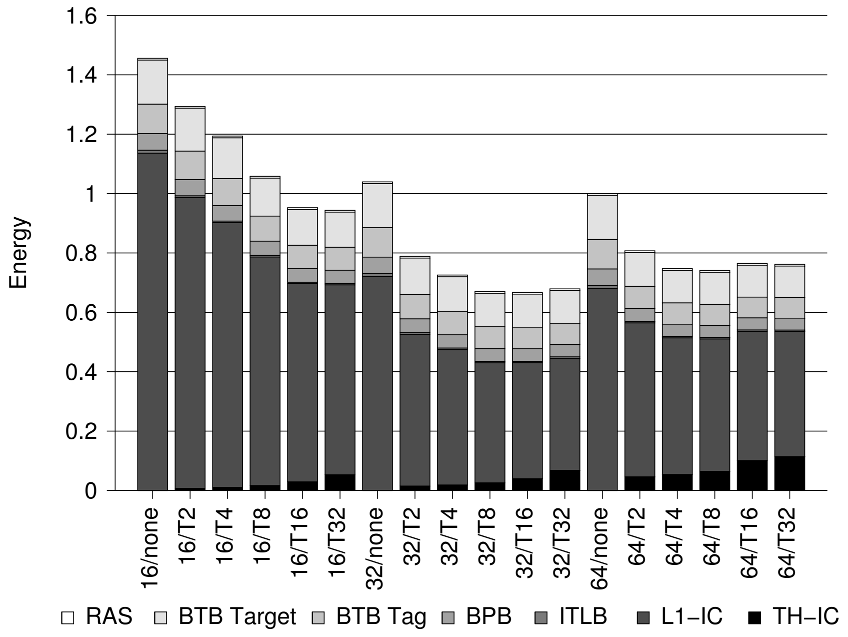


Figure 3.18: Fetch Energy, 4-wide Fetch, 25% Leakage

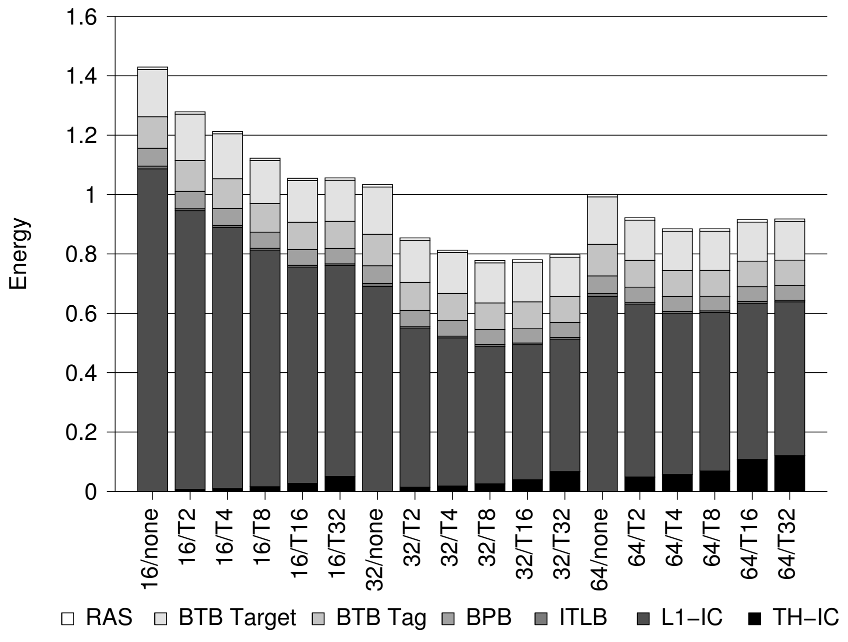


Figure 3.19: Fetch Energy, 4-wide Fetch, 40% Leakage

Benchmarks with small, tight, and easily predicted innermost loops, such as *pgp*, *sha*, *adpcm*, and *crc* show the best performance. This is due to the high TH-IC hit rate, and optimal branch predictor performance while executing these loops. Benchmarks such as *qsort*, which have higher indirect jump counts, have higher L1-IC access rates, as the targets of indirect jumps cannot be fetched from the TH-IC, and cannot be predicted.

Figure 3.21 shows the resulting I-TLB accesses. The original TH-IC alone already results in significant reductions in I-TLB accesses (in fact, exactly at the rate for L1-IC accesses), but the techniques presented in this paper further reduce the access frequency down to less than 6% of cycles for 1-wide fetch, less than 9% of cycles for 2-wide fetch, and less than 12% of cycles for 4-wide fetch. Much of the benefit comes from detecting when the current line is not the last line in the page during sequential fetches. However, some of the benefit is obtained by the use of the SP bit for direct ToCs, which allows ToCs that leave their page to be detected. The differences in I-TLB access rates between the benchmarks occur for reasons similar to the differences in L1-IC access rates.

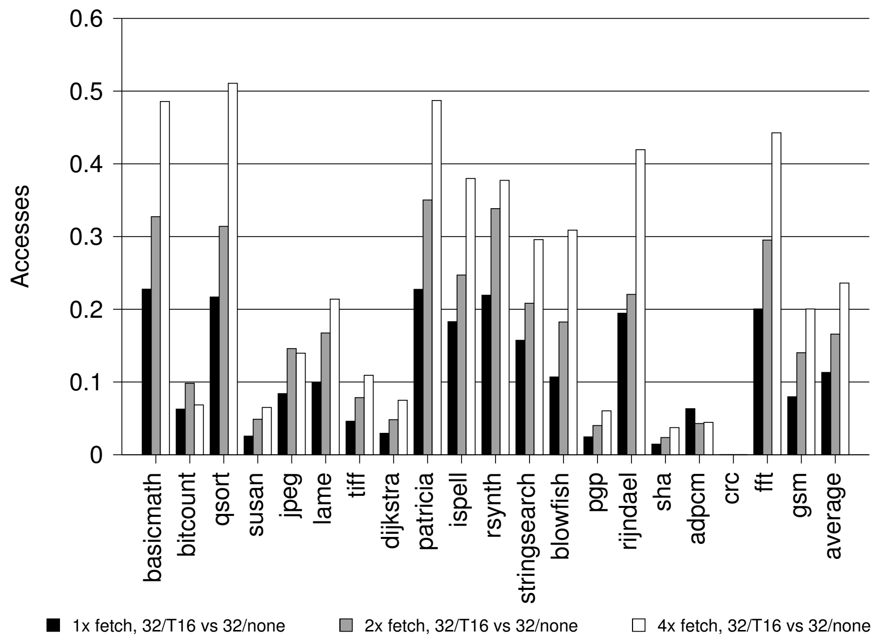


Figure 3.20: L1-IC Accesses (Normalized)

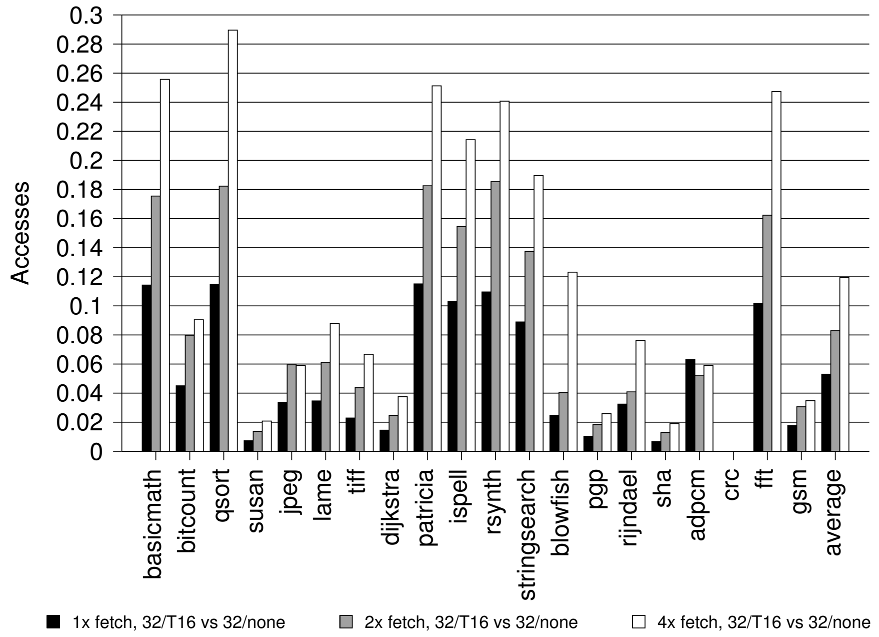


Figure 3.21: I-TLB Accesses (Normalized)

Figure 3.22 shows the resulting BPB and BTB target array accesses. Figure 3.23 shows the resulting BTB tag array accesses. The BTB tag access rate is lower than the BPB/BTB target rate because the tag array is only accessed when the branch instruction's NT bit is not set, whereas the target is accessed for all direct branches. We are able to prevent accesses to the BPB and BTB target for at least 77% of instructions for 1-wide fetch, 72% of instructions for 2-wide fetch, and 64% of instructions for 4-wide fetch. We are able to reduce BTB tag array accesses even further, to 83% of instructions for 1-wide fetch, 78% of instructions for 2-wide fetch, and to 68% of instructions for 4-wide fetch. In the MiBench suite, only 14.2% of instructions are direct transfers of control (jumps and branches). In the 1-wide fetch configuration, we are able to come to within 9% of this optimum for BPB and BTB target array accesses, and to within 3% of the optimum for BTB tag array accesses.

Note that, on average, access rates for all the fetch related structures improve when higher fetch widths are used. This is primarily due to the larger L1-IC sizes (and thus higher hit rates) that are present on the more aggressive pipeline models. In our model, L1-IC misses result in reactivation of all structures once the line fill completes, and so higher L1-IC miss rates result in higher energy.

Figure 3.24 shows the number of execution cycles for each of the simulated benchmarks, normalized against the configuration with no TH-IC (but with the same L1-IC line size). The slight improvement in performance shown here can be explained by the reduction in BTB contention as discussed in 3.4.6, and as evidenced by the corresponding reductions in BTB replacements (Figure 3.25) and branch mispredictions (Figure 3.26). Though the frequency of BTB replacements were decreased by similar fractions for all 3 pipelines, the effect on mispredictions is greater on the single-fetch pipeline because its BTB has fewer entries. Note that this improvement may not hold when changing the L1-IC line size, due to different miss rates.

Finally, Figures 3.27-3.35 show the energy usage breakdown for each benchmark on the lowest energy TH-IC-based configuration of those we tested. Since the goal is to minimize total fetch energy usage, whether or not a TH-IC is used, we have chosen to normalize each graph to the optimum L1-IC-only configuration. Thus, each graph shows the ratio of energy savings of the best TH-IC-based configuration versus the best L1-IC-only configuration. This shows that, in all cases,

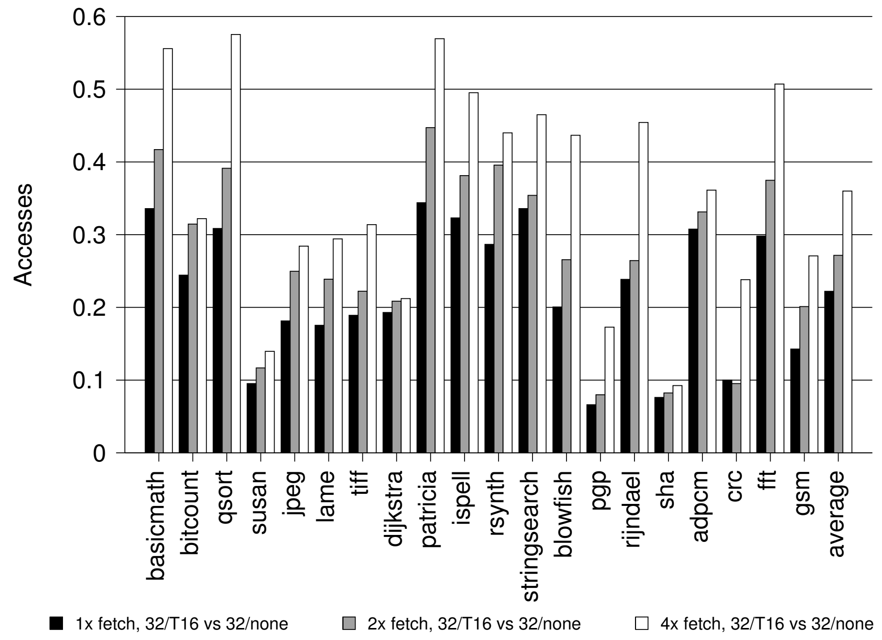


Figure 3.22: BPB/BTB Target Accesses (Normalized)

adding an optimized TH-IC provides benefit over simply optimizing the L1-IC and not using a TH-IC.

3.6 Hardware Implementation and Analysis

In order to more accurately model the power and performance of the TH-IC, it has been implemented in synthesizable RTL on the OpenRISC 5-stage pipeline described above. The RTL model was synthesized using a low-power 65 nm technology by ST Microelectronics. In this hardware implementation, only the basic TH-IC with NS, NT, and TL bits has been implemented.

Bardizbanyan et al[4] shows that requiring that a complete block be transferred between caches in a single cycle incurs tremendous increases to access energy. Implementing such a transfer requires that the caches have ports as wide as a full cache block. In addition, for associative caches, all ways of the cache must be read simultaneously, as the tag array is typically accessed in parallel with the data array, with the tag check and way selection occurring in the same cycle. For a 4-way

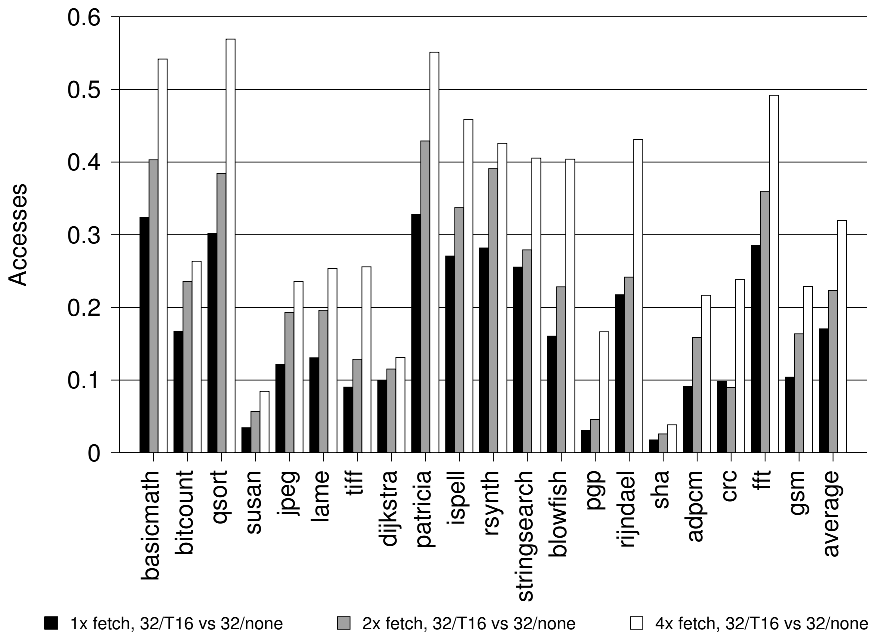


Figure 3.23: BTB Tag Accesses (Normalized)

set associative cache with 16-byte blocks, this requires what is essentially a 512-bit port.

For these reasons, the TH-IC has been implemented to transfer the cache block over multiple cycles. However, instructions can be forwarded to the pipeline as they are read from the L1-IC, as long as the pipeline is requesting the instruction that is currently being transferred. This is the case as long as the instructions are requested in consecutive sequential order without moving to a different line, and the pipeline does not stall between consecutive instructions. For example, when transferring the first instruction of a block, it will be forwarded to the pipeline. If the next instruction requested by the pipeline is at the next consecutively sequential address and is also within the same block, it can also be automatically forwarded. However, since the fill occurs over consecutive cycles, if the pipeline stalls during the fill operation, the cache cannot continue forwarding filled instructions, and must wait until the fill operation completes to satisfy the next pipeline request.

The downside to this strategy is that each TH-IC miss imposes a penalty of one cycle for each

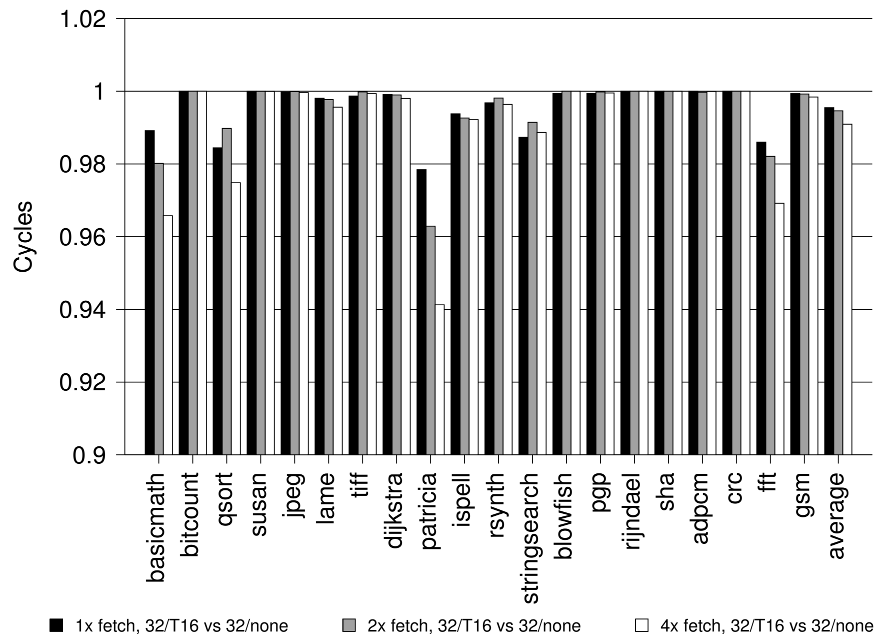


Figure 3.24: Simulated Cycles (Normalized)

remaining instruction in the cache block. However, this penalty is smaller than the miss penalty for a filter cache, because the TH-IC miss is detected ahead of time. On a true TH-IC miss, the transfer of instructions from the L1-IC to the TH-IC is immediately initiated with the first L1-IC access in the block. With a traditional filter cache, the filter cache must be accessed before the fill is initiated, imposing an additional 1-cycle penalty. So the performance of this TH-IC is lower than that of an L1-IC on its own, but still better than that of a traditional filter cache.

Table 3.3 shows the area used by instruction fetch units both with and without a TH-IC. Due to the limited selection of block memories that were available, the L1-IC valid bits and LRU table were implemented using register cells. As a result, these components occupy a disproportionate amount of area, and do not have chip-select signals that can be used to reduce their power consumption. The TH-IC adds small memories for the TH-IC instruction data and the ID field. There are very few valid and NS bits (32 each in this implementation), and the use of the NT and TL bits prevents them from being implemented as block memories. These bits are instead implemented using register

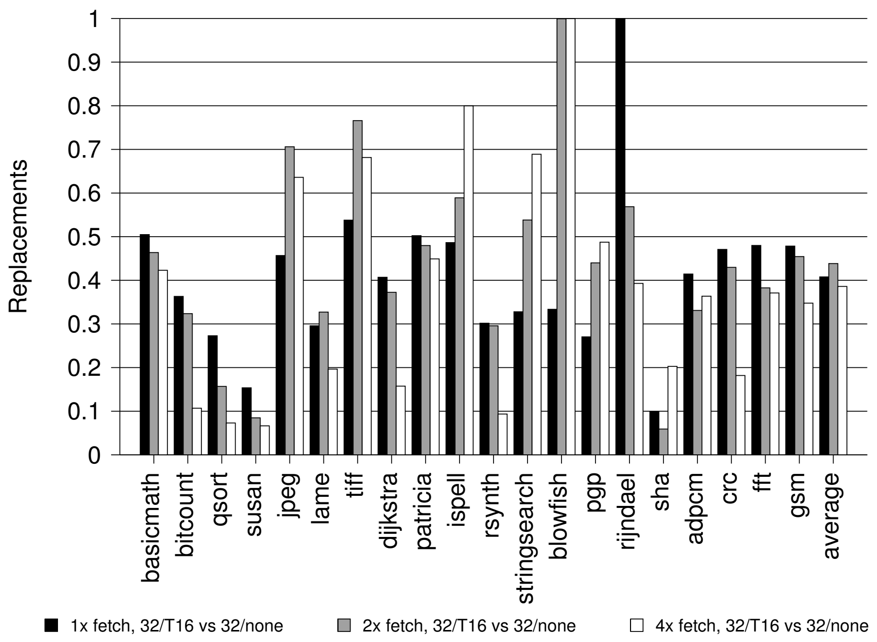


Figure 3.25: BTB Replacements (Normalized)

cells in the datapath subcomponent, thus accounting for its increase in size. Thic TH-IC adds under 9% to the total area of the instruction fetch unit.

3.7 Experimental Results

Figure 3.36 shows the performance impact of introducing the TH-IC implemented as just described. The average performance reduction for this implementation is less than 2%.

Figure 3.37 shows the relative outcomes of fetches for pipelines with and without a TH-IC. The bars whose labels begin with "L1-Only" correspond to the pipeline without a TH-IC, and those whose labels begin with "TH-IC" correspond to a pipeline with a TH-IC. The outcomes for the L1-only configuration are broken down as follows:

- **L1-Only Filled**: instructions forwarded to the pipeline during block fills;
- **L1-Only Hits**: cache hits;

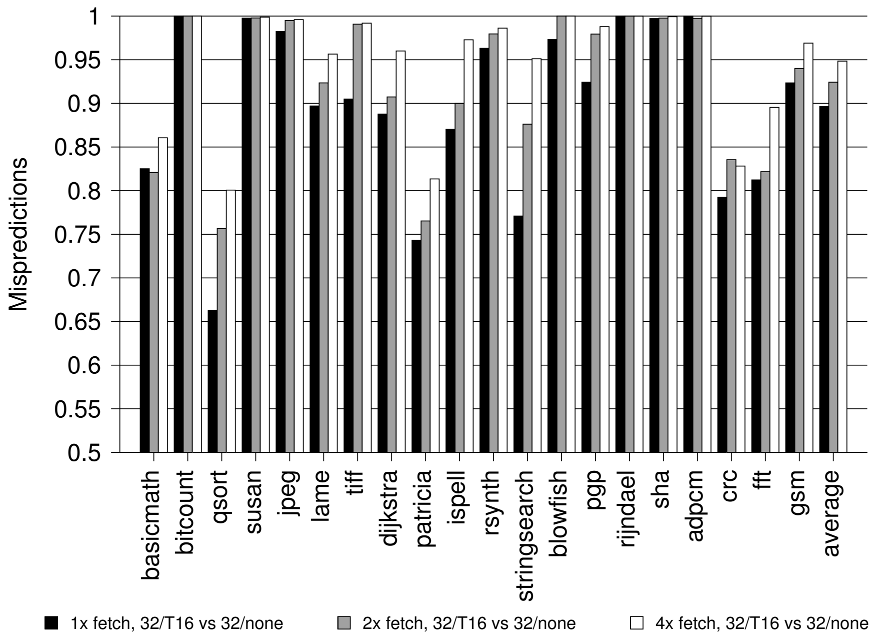


Figure 3.26: ToC Mispredictions (Normalized)

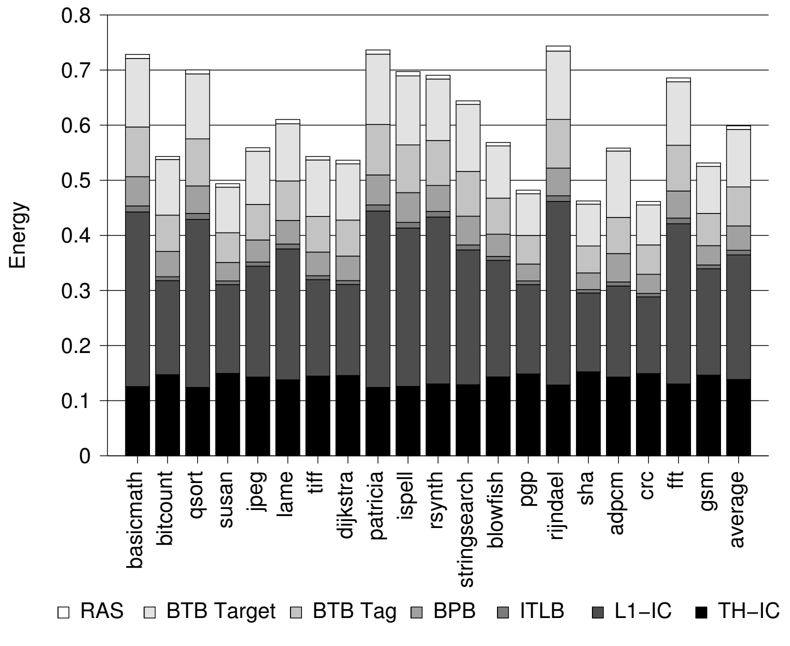


Figure 3.27: Fetch Energy, 1-wide Fetch, 10% Leakage, 32/T8 vs. 64/none

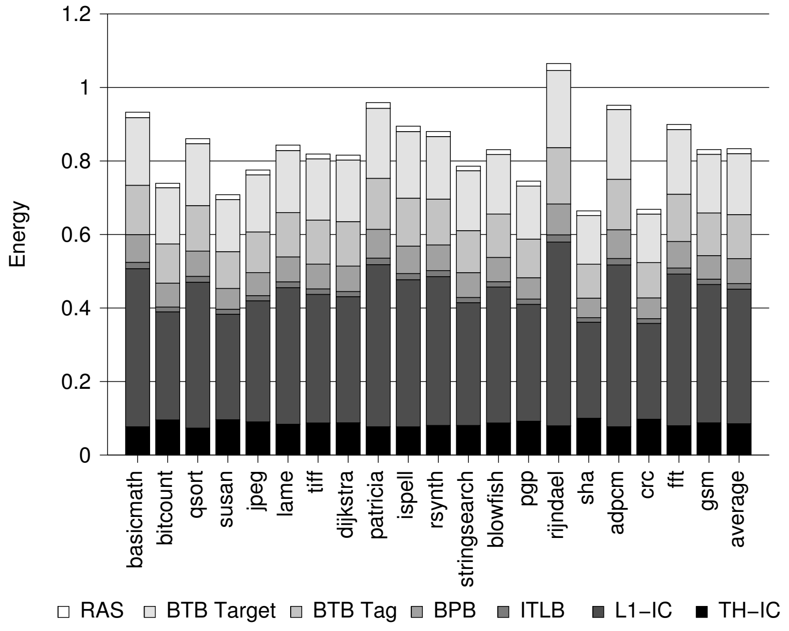


Figure 3.28: Fetch Energy, 1-wide Fetch, 25% Leakage, 16/T8 vs. 64/none

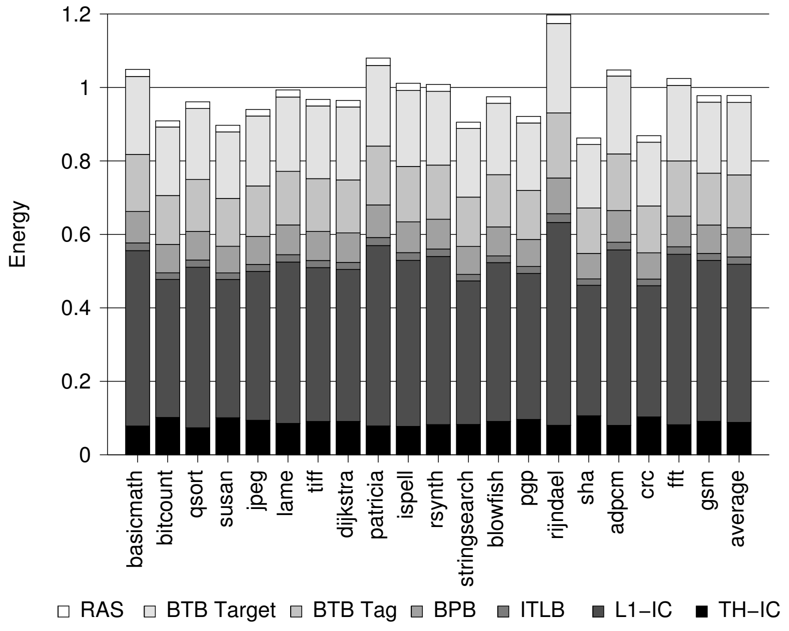


Figure 3.29: Fetch Energy, 1-wide Fetch, 40% Leakage, 16/T4 vs. 64/none

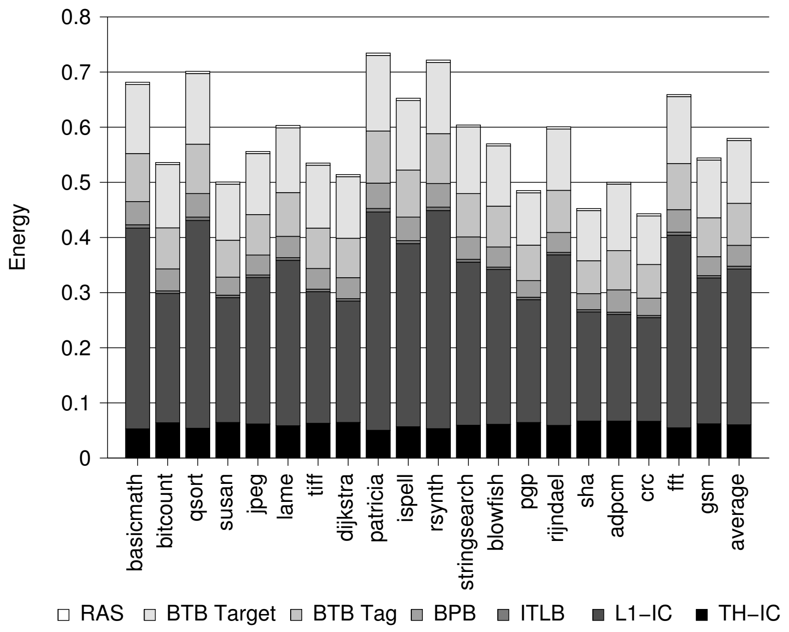


Figure 3.30: Fetch Energy, 2-wide Fetch, 10% Leakage, 32/T8 vs. 64/none

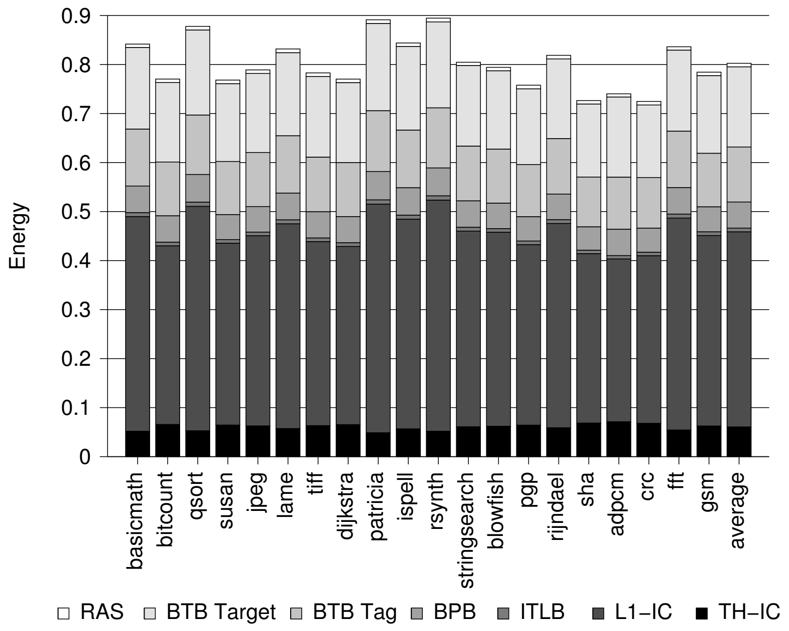


Figure 3.31: Fetch Energy, 2-wide Fetch, 25% Leakage, 32/T8 vs. 64/none

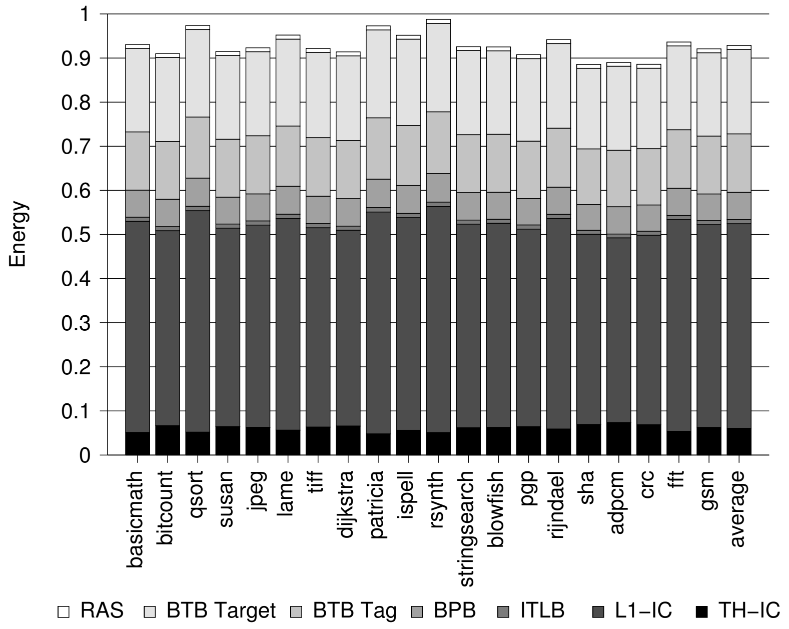


Figure 3.32: Fetch Energy, 2-wide Fetch, 40% Leakage, 32/T8 vs. 64/none

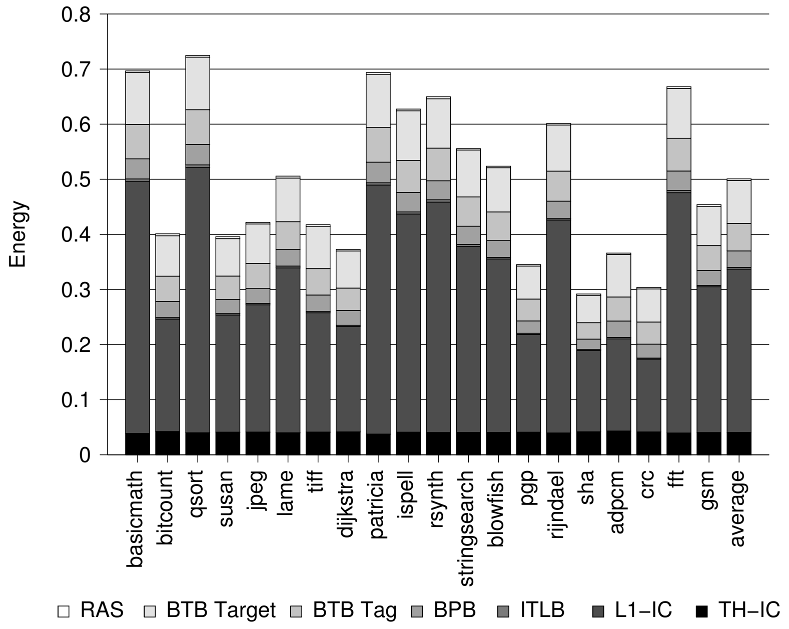


Figure 3.33: Fetch Energy, 4-wide Fetch, 10% Leakage, 32/T16 vs. 32/none

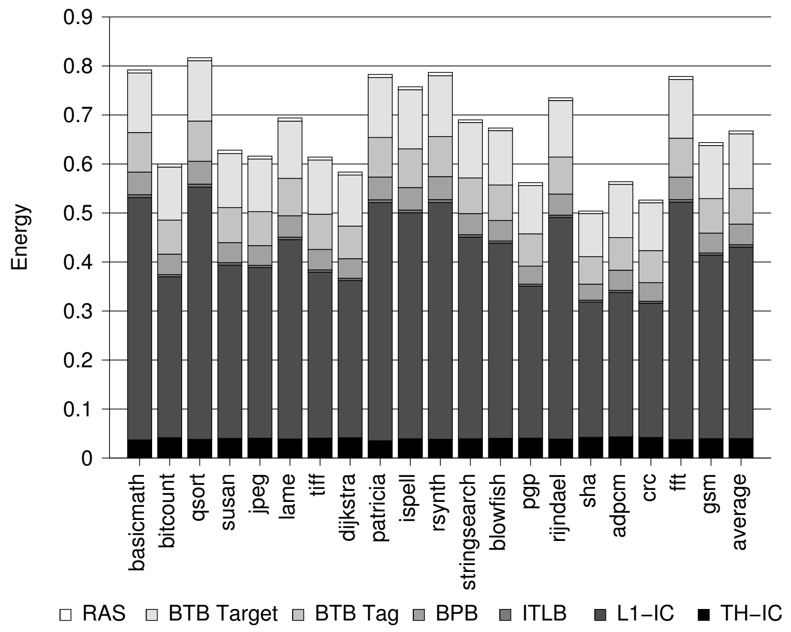


Figure 3.34: Fetch Energy, 4-wide Fetch, 25% Leakage, 32/T16 vs. 32/none

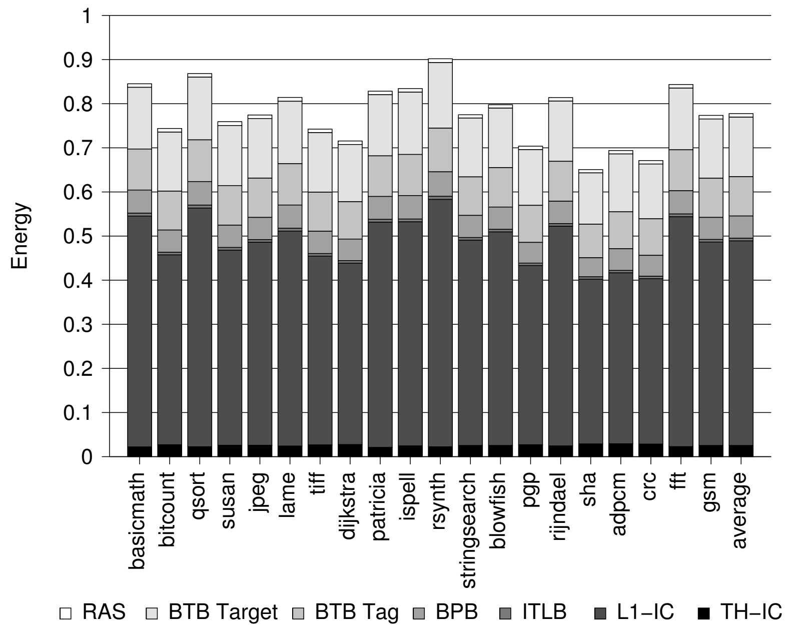


Figure 3.35: Fetch Energy, 4-wide Fetch, 40% Leakage, 32/T8 vs. 32/none

Table 3.3: Instruction Fetch Unit Area Results

	L1-IC Only	L1-IC + TH-IC
L1-IC Instruction Memory	290370	290370
L1-IC Tag Memory	87115	87115
L1-IC Valid Bits	31412	31637
L1-IC LRU Table	47242	47242
TH-IC Instruction Memory	—	11440
TH-IC ID Memory	—	8899
Control	1234	1540
Datapath	2033	21026
Total Area	459406	499271

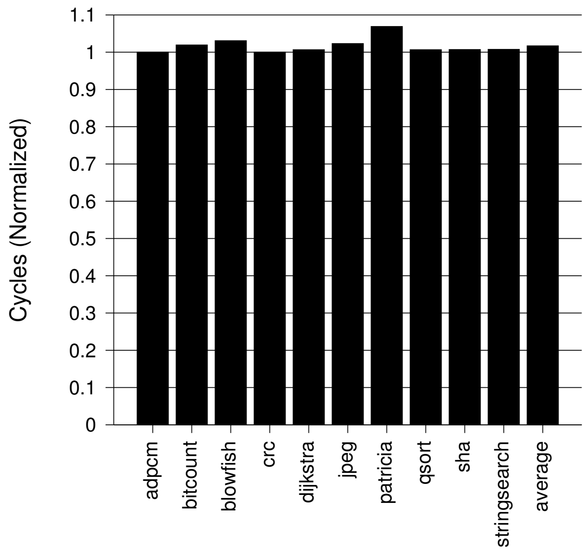


Figure 3.36: TH-IC Performance Impact

- **L1-Only Tagless**: cache hits that could be performed without a tag check (sequential fetches from within the same cache block);

The outcomes for the TH-IC configuration are broken down as follows:

- **TH-IC L1 Filled**: instructions forwarded to the pipeline during block fills for L1-IC misses; i.e. both L1-IC and TH-IC are being filled;
- **TH-IC Filled**: instructions forwarded to the pipeline during block fills for TH-IC-only misses; i.e. the TH-IC is being filled from the L1-IC;
- **TH-IC True Misses**: instructions fetched from L1-IC, true TH-IC miss
- **TH-IC False Misses**: instructions fetched from L1-IC, false TH-IC miss
- **TH-IC Hits**: instructions fetched from TH-IC

As shown in this figure, the TH-IC is able to achieve a reasonably good hit rate of more than 87%. The **TH-IC True Misses** category is quite small, at less than 4%. The additional latencies imposed by the TH-IC only occur for this category of fetches, and the **TH-IC Filled** category represents fetches that prevented some of this latency. On average, fewer than 7% of fetches were satisfied in this way, however for some benchmarks (patricia, jpeg) the fraction is significant.

Due to additional latencies introduced by the TH-IC, there is a slight decrease in the number of fetch requests. This occurs because the latencies prevent the pipeline from speculating quite as far into the instruction stream, and thus fewer misspeculations can occur.

Figure 3.38 compares the absolute power usage of the instruction fetch units, for both the L1-only configuration and the TH-IC configuration. The power usage for the L1-Only configuration is broken down as follows:

- **L1-Only Inst SRAM**: power used by instruction memory
- **L1-Only Tag SRAM**: power used by tag memory
- **L1-Only Valid Bits**: power used by valid bits

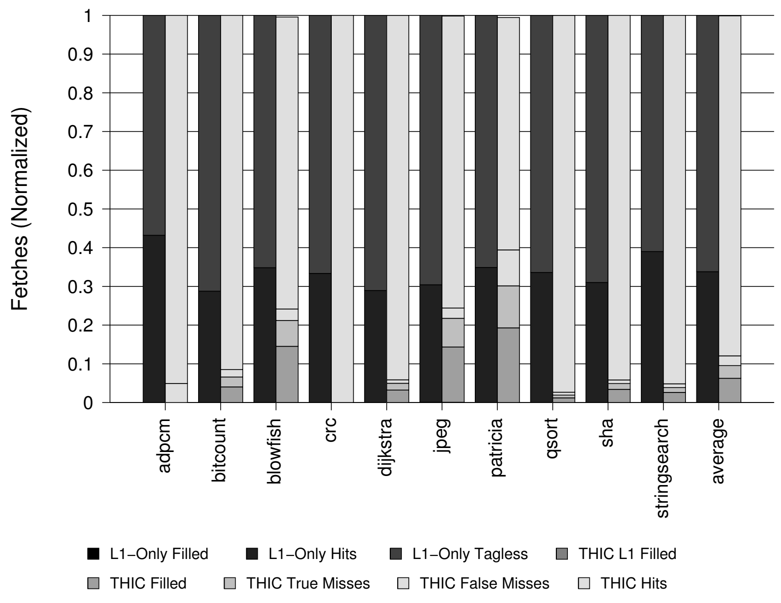


Figure 3.37: TH-IC Fetches

- **L1-Only LRU Table**: power used by LRU table
- **L1-Only Datapath**: power used by datapath (multiplexers for address and instruction)
- **L1-Only Control**: power used by control logic

The power usage for the TH-IC configuration is broken down as follows:

- **TH-IC L1 Inst SRAM**: power used by the L1-IC instruction memory
- **TH-IC Tag SRAM**: power used by L1-IC tag memory
- **TH-IC Valid Bits**: power used by L1-IC valid bits
- **TH-IC LRU Table**: power used by L1-IC LRU table
- **TH-IC Inst SRAM**: power used by TH-IC instruction memory
- **TH-IC ID SRAM**: power used by TH-IC ID memory

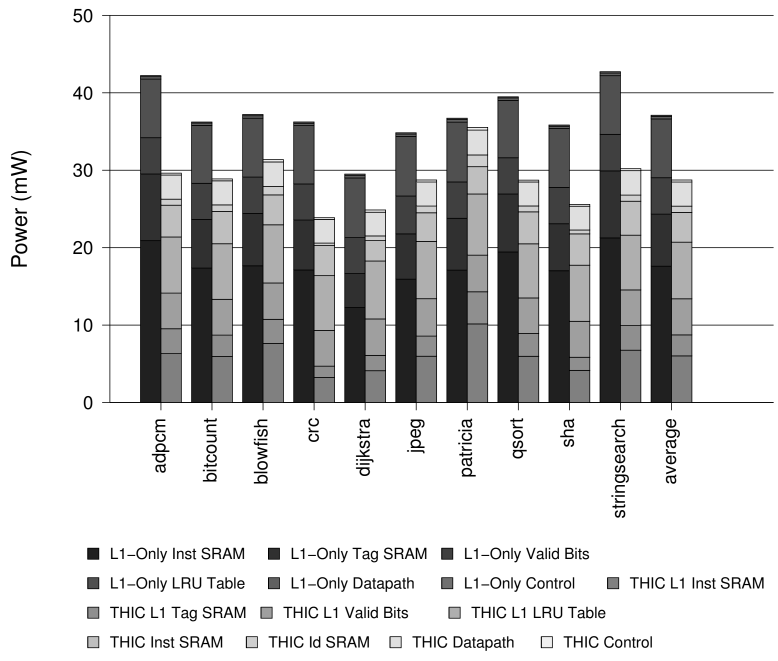


Figure 3.38: TH-IC Instruction Fetch Power Usage

- **TH-IC Datapath**: power used by datapath (multiplexer for address and instruction; TH-IC Valid, NS, NT, TL bits)
- **TH-IC Control**: power used by control logic

As shown in this figure, although the TH-IC introduces new memories and significant new datapath logic, it manages a substantial reduction in power usage. This occurs primarily through a reduction in the access rates of the L1 instruction and tag memories. Introducing the TH-IC results in an instruction fetch power reduction of 22%.

Figure 3.39 compares the relative instruction fetch energy usage of each configuration. Although the TH-IC results a small performance penalty of less than 2%, this penalty is far less than the improvement in power usage, and thus the TH-IC reduces instruction fetch energy over 20%.

Figure 3.40 shows the total core power usage. As this figure shows, instruction fetch is the dominant contributor to total core power. This occurs due to the relatively large instruction cache size, combined with its frequent, nearly continuous access rate. In comparison, although the data

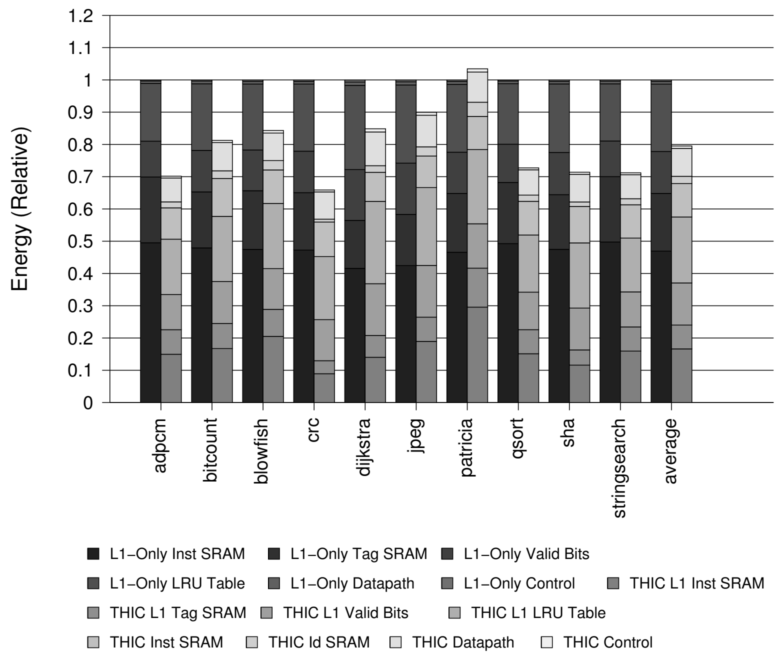


Figure 3.39: TH-IC Instruction Fetch Energy Usage

cache is equal in size to the instruction cache, and involves a more complex datapath and control logic, it is accessed much less frequently and thus has lower power usage. The overall effect is a reduction in total core power of over 9%.

Figure 3.41 shows relative total core energy usage. Although the TH-IC introduces a small performance penalty, the penalty is not sufficient to outweight the power reduction of the TH-IC. Thus total core energy is substantially reduced as well, by over 7%. However, the delays introduced when filling TH-IC lines result in smaller improvements to energy usage that was found in the higher-level simulations.

When the TH-IC was first introduced[14], high level simulations produced an estimated energy reduction of nearly 30% for a configuration similar to the one presented here. That work used a slightly smaller 16KB L1-IC, while this work used a 32KB L1-IC, and it should be expected that the power savings would be larger with the larger L1-IC. However, several inaccuracies in the high level simulation resulted in an overestimation of the power savings that could be achieved. While

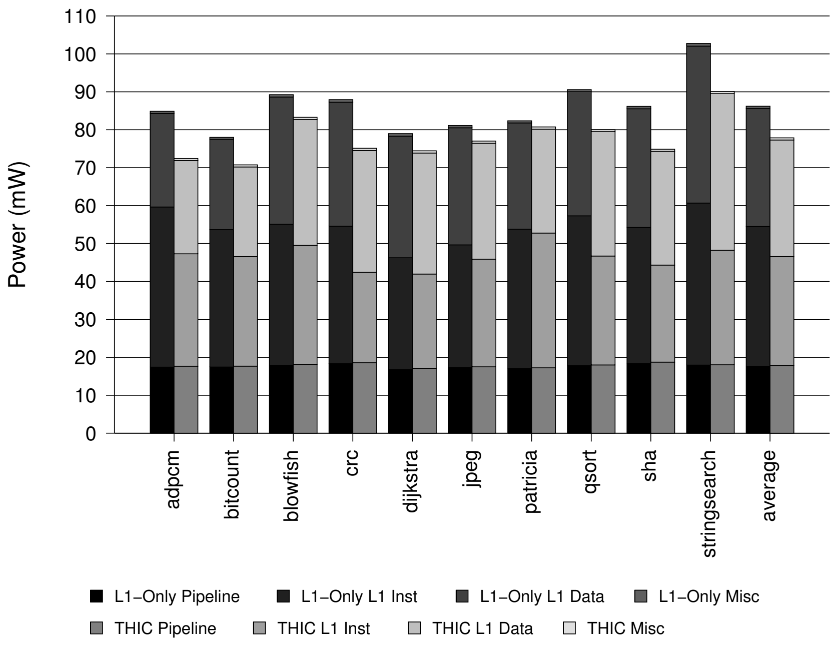


Figure 3.40: TH-IC Total Core Power Usage

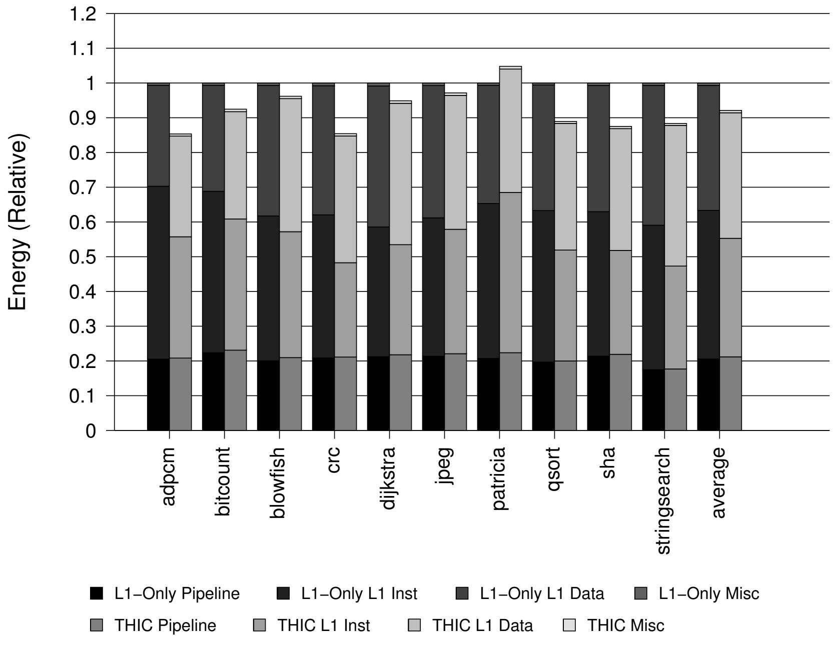


Figure 3.41: TH-IC Total Core Energy Usage

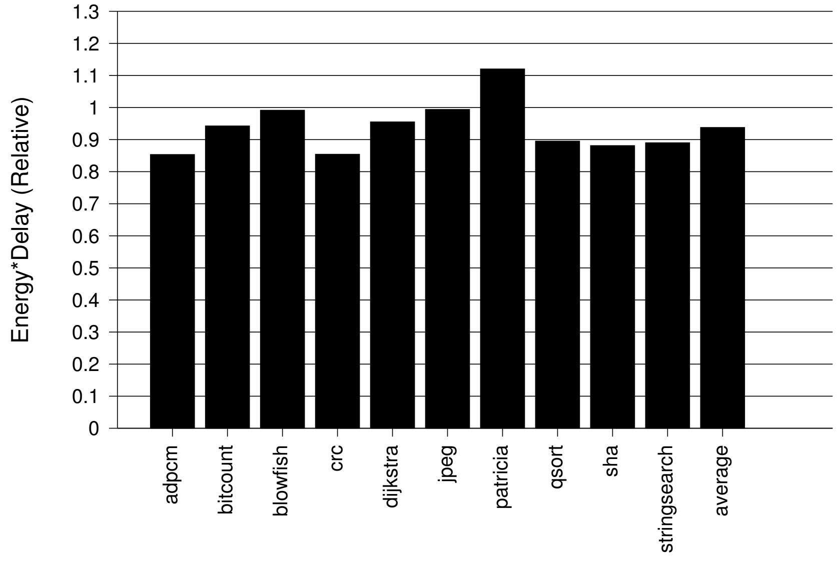


Figure 3.42: TH-IC Core Energy*Delay

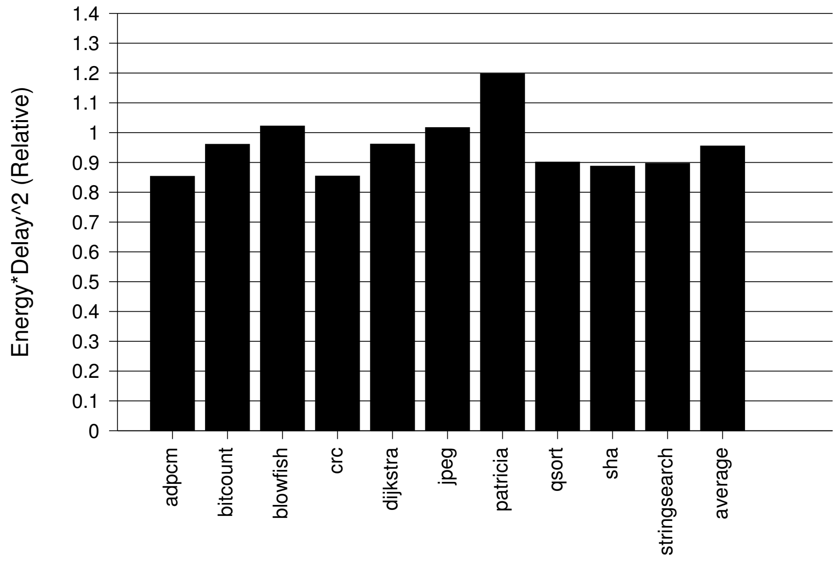


Figure 3.43: TH-IC Core Energy*Delay^2

the results of this study affirms that the TH-IC is capable of reducing power and energy usage, it also shows the need for improved high-level modelling techniques that more accurately predict the energy usage of new architectures.

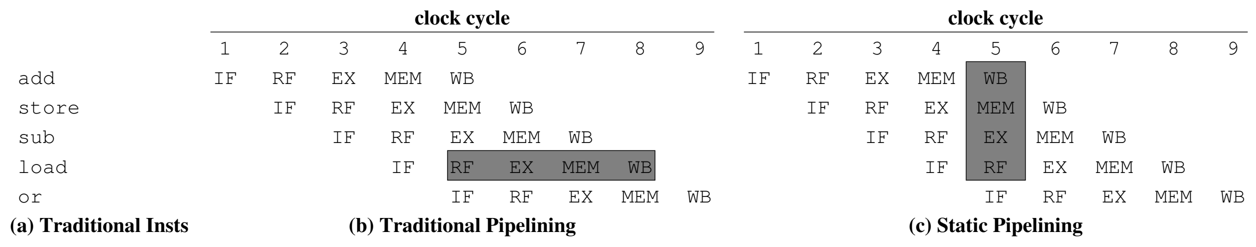
CHAPTER 4

STATIC PIPELINING

The second architectural technique to be researched in this dissertation is called static pipelining. This chapter provides background information on static pipelining.

Static pipelining is a reconceptualization of the traditional 5-stage RISC pipeline that replaces the complex, dynamically controlled instruction flow found in these pipelines with simple, statically determined control generated by the compiler. As shown in Figure 4.1, in a traditional 5-stage RISC pipeline, each instruction is executed over 5 sequential stages, each stage taking (at least) one cycle to complete. These stages are *fetch* (IF), *decode* (ID), *execute* (EX), *memory* (MEM), and *writeback* (WB). A different instruction occupies each stage of the pipeline, and instructions flow down the pipeline, starting at IF and committing at WB. Each instruction encodes a simple operation, such as an ALU operation, a transfer of control, or a memory access.

Figure 2.2 shows the layout of an OpenRISC 5-stage pipeline. Instructions are fetched from the L1 cache in the IF stage. In the ID stage, the instruction is decoded, and additionally, the register file is read. ALU operations are performed in the EX stage, and memory accesses are performed in the MEM stage. Finally, in the WB stage, values are written the the register file, and exceptions may be handled. Each stage is separated from the one that follows by a group of pipeline registers, which are written every cycle (ignoring stalls and flushes).



(a) Traditional Insts	1	2	3	4	5	6	7	8	9
add	IF	RF	EX	MEM	WB				
store		IF	RF	EX	MEM	WB			
sub			IF	RF	EX	MEM	WB		
load				IF	RF	EX	MEM	WB	
or					IF	RF	EX	MEM	WB

(b) Traditional Pipelining (clock cycle)

	1	2	3	4	5	6	7	8	9
add	IF	RF	EX	MEM	WB				
store		IF	RF	EX	MEM	WB			
sub			IF	RF	EX	MEM	WB		
load				IF	RF	EX	MEM	WB	
or					IF	RF	EX	MEM	WB

(c) Static Pipelining (clock cycle)

Figure 4.1: Traditional Pipelining vs. Static Pipelining

While the traditional processor dynamically pipelines the steps required to perform each instruction, the static pipeline instead relies on the compiler to simulate the pipelining. The effects described above (register read/write, functional unit operations, memory accesses, and pipeline register writes) are exposed directly to the compiler. The compiler breaks each traditional instruction into equivalent static pipeline effects. The compiler then schedules and parallelizes effects, within the constraints of the static pipeline microarchitecture.

The compiler generates code by first compiling to a standard RISC-like ISA. It then expands each RISC instruction into a number of effects that mimic the flow of the instruction through a standard RISC pipeline. For example, the standard RISC instruction

$\mathsf{add}\ r1, r2, r3$

would be expanded into the sequence

$RS1 = r2$

$RS2 = r3$

$OPER1 = RS1 + RS2$

$r4 = OPER1$

The expanded code is then optimized using standard optimization techniques, such as copy propagation, dead code elimination, code motion, and so on. Once compilation is finished, effects are parallelized as much as the dependencies between them and the available templates allow.

In the static pipeline, nearly all of the dynamic control logic, such as hazard detection, forwarding, is eliminated, reducing circuit area and thus power. The pipeline registers between pipeline stages are replaced with compiler-controlled temporary registers, and forwarding of values is handled by the compiler using these registers. In instances where forwarded values would normally be written to the register file, but never reused, the write to the register file can be eliminated, again reducing power. The performance of such a pipeline is tremendously compiler dependent, and reliant on a number of advanced optimization techniques. However, an in-house compiler is able to attain cycle counts comparable to that of a traditional RISC pipeline for many benchmarks.

Figure 4.2 shows the microarchitecture of a static pipeline based on a standard 5-stage RISC architecture. The static pipeline has all the major components of a standard RISC pipeline, such as a register file, ALUs, a load/store unit, and a branch predictor. As mentioned, the pipeline registers are replaced with compiler-controlled temporary registers. These registers are not arbitrarily addressible, the way register file registers are. Each register can only be written with values from a small number of sources. These registers are:

- RS1, RS2 (register sources 1 & 2): can be written with a entry read from the register file
- LV (load value): can be written with a value read from memory
- OPER1 (operand 1): written with the value produced by the first functional unit (FU1)
- OPER2 (operand 2): written with the value produced by the second functional unit (FU2)
- SE: can be written with the (possibly sign extended) immediate value from an instruction
- CP1, CP2: temporary "copy" registers; each can be written with a value from most of the other temporary registers
- ADDR: stores an address that can be used for memory accesses and control transfers
- SEQ: can be written with the value of PC + 1

The static pipeline instruction provides a number of independent effects that occur in parallel, as dictated by the so-called "wide" instruction format shown in Table 4.1. Most of these effects are essentially multiplexer selectors and register enables that choose from where each of these registers will take their next values. The supported effects are:

- FU1 (functional unit 1): performs basic integer single-cycle ALU operations
- FU2 (functional unit 2): performs long-latency integer and floating-point operations
- CP (copy): performs a write into one of the copy registers (CP1 or CP2)

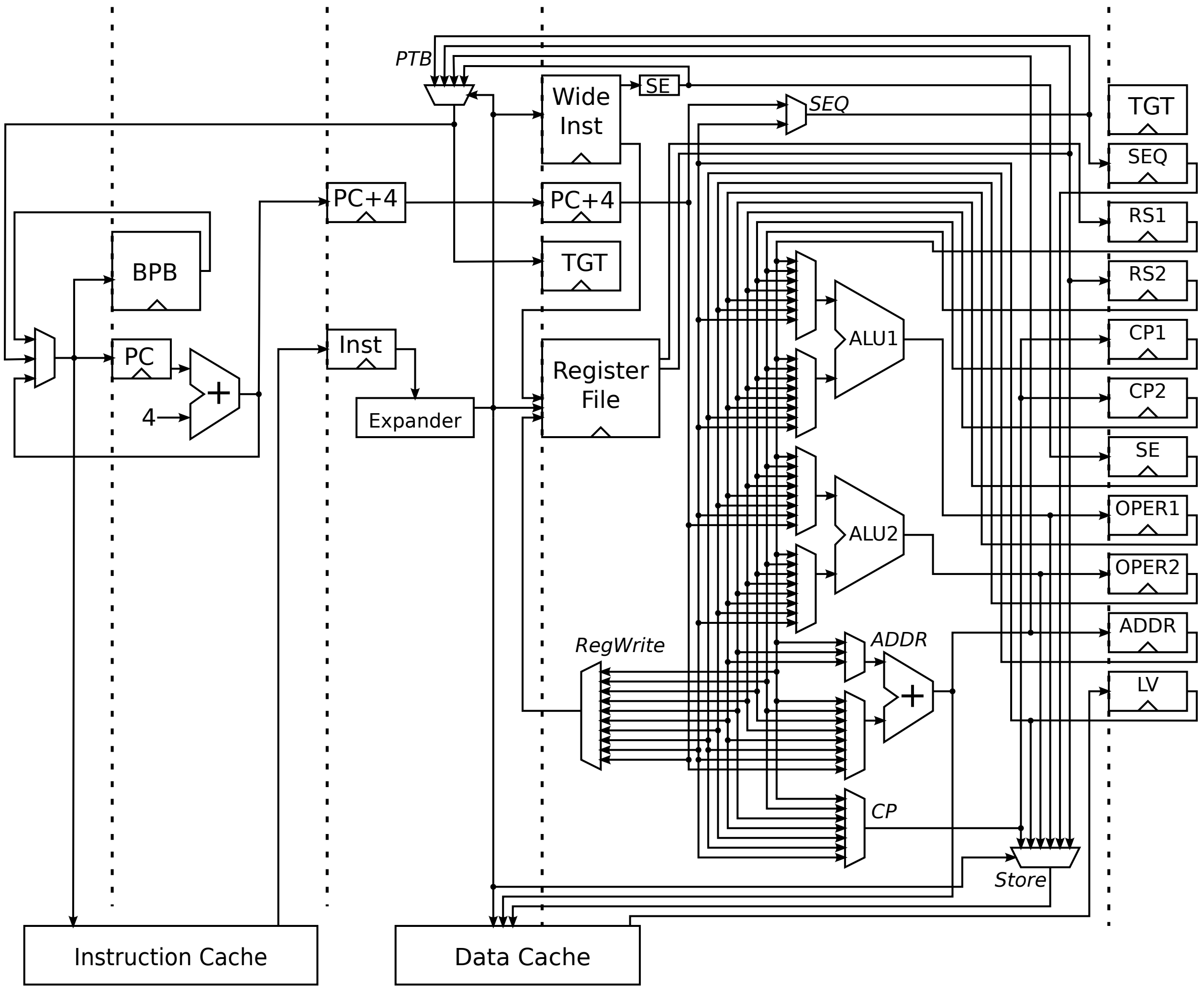


Figure 4.2: Static Pipeline Block Diagram

- SEQ (sequential address register): performs a write into the sequential address register; used primarily to save the PC upon entering a loop
- PTB (prepare-to-branch): initiates a transfer of control
- RF (register file): controls reading and writing the register file
- SE (sign-extend): controls sign-extension and handling of immediate values
- MEM (memory): controls load/store operations
- ADDR (address): supplies address for load/store operations and control transfers

The static pipeline exposes more independent effects to the compiler than a typical processor. As a result, the full instruction is relatively large, at over 80 bits wide. The compiler typically cannot parallelize more than a handful of effects per instruction, however, so most instructions have a no-op for several of the effects. In order to reduce the size of each instruction, a set of 32 "templates" are chosen, as shown in Figure 4.3. The most significant five bits of the 32-bit instruction are used to identify which template is in use. The remaining bits are divided into fields of various widths, each of which can be used to encode a single effect. The 5-bit template identifier determines which combination of fields will be used, as well as which effect each field encodes. The effects that can be encoded by a field are limited by the width of the field. For a given template identifier, the wide instruction can be generated by extracing each field in the 32-bit instruction to the appropriate effect in the wide instruction. The set of templates that are used to compress instructions is determined by profiling.

4.1 Compilation

As mentioned earlier, the static pipeline compilation process is more complex than for a traditional pipeline. Figure 4.4 shows the flow for our static pipeline compiler. First, a standard C compiler front-end (LCC) is used to compile C code to traditional MIPS instructions. These RTLs are then optimized using a traditional code optimizer (VPO), to produce the optimized MIPS code.

Table 4.1: Static Pipeline Wide Instruction Format

Effect	**Field**	**Encoding**	**Effect**	**Field**	**Encoding**	**Effect**	**Field**	**Encoding**
FU1	OP	NOP	FU2	OP	NOP	MEM	OP	NOP
		ADD			ADD			STORE
		SUB			ADDF			LOAD
		AND			SUBF			LOADU
		OR			MULT		SIZE	BYTE
		XOR			MULTF			HALF
		NOT			DIV			WORD
		SLL			DIVU			CACHE
		SRL			SLTF		VALUE	RS1
		SRA			ITF			RS2
		SLT			ITD			OPER1
		SLTU			FTI			CP1
		BEQ			FTD			OPER2
		BNE			DTI			LV
	SRC1	OPER1			DTF			SEQ
		OPER2		SRC1	OPER1			CP2
		RS1			OPER2	RF	DESTEN	0/1
		RS2			RS1		DESTVAL	OPER1
		LV			RS2			OPER2
		CP1			LV			RS1
		CP2			CP1			RS2
		ZERO			CP2			LV
	SRC2	OPER1			ZERO			CP1
		OPER2		SRC2	OPER1			CP2
		RS1			OPER2			SE
		RS2			RS1		DEST	5 bits
		LV			RS2		SRC1EN	0/1
		CP1			LV		SRC1	5 bits
		CP2			CP1		SRC2EN	0/1
		SE			CP2		SRC2	5 bits
CP	EN	0/1			SE	SE	EN	0/1
	SRC	RS1	SEQ	EN	0/1		SRC	NOP
		RS2		SRC	PCINCR			CP1
		OPER1			LV			CP2
		OPER2	PTB	EN	0/1			IMM
		SE		COND	0/1		HI	0/1
		LV		SRC	RS2		IMM	16 bits
		ADDR			SE			
	DEST	CP1			SEQ			
		CP2			ADDR			

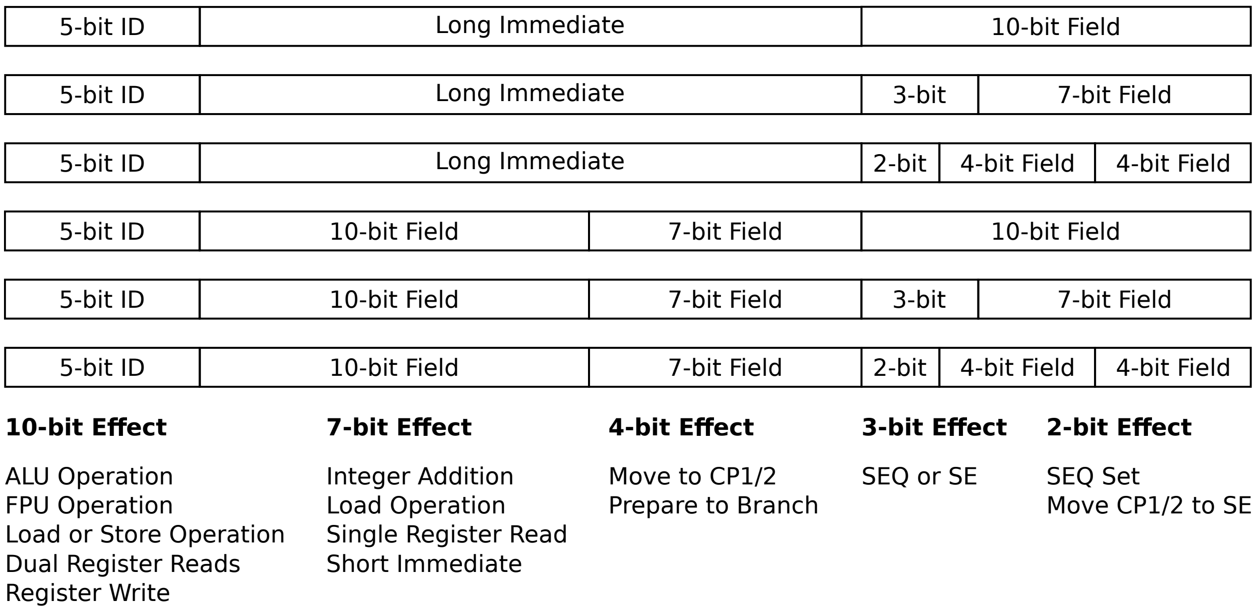

5-bit ID | Long Immediate | 10-bit Field
5-bit ID | Long Immediate | 3-bit | 7-bit Field
5-bit ID | Long Immediate | 2-bit | 4-bit Field | 4-bit Field
5-bit ID | 10-bit Field | 7-bit Field | 10-bit Field
5-bit ID | 10-bit Field | 7-bit Field | 3-bit | 7-bit Field
5-bit ID | 10-bit Field | 7-bit Field | 2-bit | 4-bit Field | 4-bit Field
10-bit Effect
ALU Operation
FPU Operation
Load or Store Operation
Dual Register Reads
Register Write
7-bit Effect
Integer Addition
Load Operation
Single Register Read
Short Immediate
4-bit Effect
Move to CP1/2
Prepare to Branch
3-bit Effect
SEQ or SE
2-bit Effect
SEQ Set
Move CP1/2 to SE

Figure 4.3: Static Pipeline Templated Instruction Formats

Each MIPS instruction in the optimized MIPS code is then expanded to a set of equivalent static pipeline effects. These effects are then further optimized using a version of VPO that has been modified to process static pipeline code. Thus the code is essentially optimized twice, first as MIPS code, and then as static pipeline code.

Although the MIPS instructions supplied to the code expander are already optimized, the code expander causes a large number of redundant or otherwise optimizable effects that were not initially present in the MIPS code to be exposed. For example, the expanded code exposes

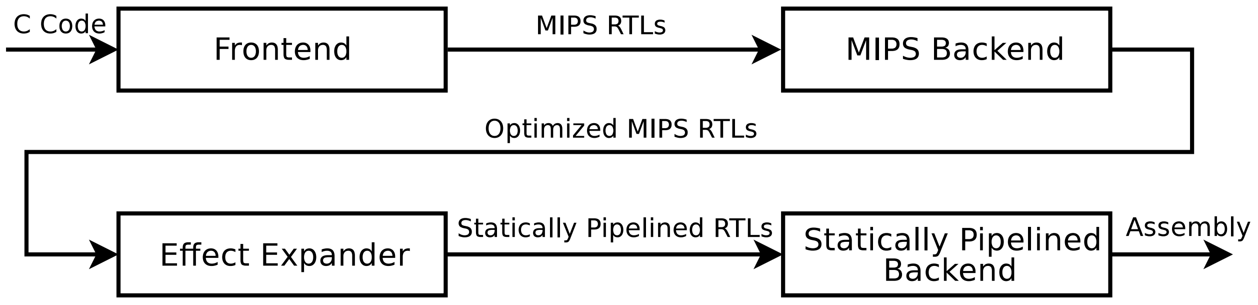


Figure 4.4: Static Pipeline Compilation Flow

accesses to the intermediate temporary registers that were not present before. The new data flow chains that are present in the static pipeline code can be optimized using traditional optimizations such as common sub-expression elimination, copy propagation, and dead assignment elimination. Additionally, operations such as branch target calculation are directly exposed to the compiler. Since, for example, the targets of branches are often loop-invariant, the code to calculate them can be hoisted out of loop bodies in order to avoid calculating them every cycle. The calculation of targets of loop branches can be optimized away entirely, by recording the PC of the first instruction in the loop just before the loop is entered, using the SEQ register. These sorts of optimizations are not at all possible on MIPS code, because MIPS code hides accesses to internal registers, while the static pipeline code exposes them.

Scheduling static pipeline code is a more difficult task for the static pipeline than traditional pipelines. Static pipeline code typically exhibits false dependences more frequently than traditional RISC code, due to the many temporary registers used. The scheduler attempts to schedule the block first by ignoring false dependences, and choosing instructions from the longest dependence chain in each block. If this fails, the schedule is not modified. However, the independence of effects within each static pipeline instructions allows increased flexibility in scheduling, and moving effects between basic blocks.

In order to reduce code size, the code must be compacted, as discussed earlier. This task is divided into two parts: choosing the set of templates supported by the processor, and then choosing which templates to use when compacting the code. In order to generate the set of supported templates, a set of benchmarks representative of the expected workload is compiled and simulated. For this initial compilation, wide instructions are used. The wide instructions allocate space for all effects, even when those effects are not in use, but only the wide instructions that can be represented using one of formats shown in Figure 4.3, but with no limitation to the effects encoded by each field. The benchmarks are then simulated, and the simulator counts the number of times each combination of effects occur while simulating each benchmark. By choosing the 32 most common combinations of effects, over 90% of all instructions are covered. This set of 32 supported templates is used to generate the final compiler, simulator, and hardware instruction expander.

4.2 Hardware Implementation and Analysis

The statically pipelined processor shown in Figure 4.2 uses a 3-stage pipeline. As in the traditional RISC pipeline, the first stage is the fetch stage. The static pipeline does not require a traditional instruction decoder, however, it does require a template expander, and this is performed in its own stage. The address and data associated with memory access instructions is sent to the data cache at the end of the instruction expansion stage. The final stage is the execute stage, in which the ALUs are activated, control transfers are resolved, and memory accesses are completed. All registers and the register file are written at the end of the execute stage.

The template expander is responsible for expanding the 32-bit instructions described in the previous section into a wide instruction containing all possible effects. As mentioned in the previous section, the set of templates are determined through profiling. Once the profiling has been completed, a description of the templates are written to a file for later processing. This file is used to generate the assembler, simulator, and the RTL code for the static pipeline.

The RTL code for the template expander is generated by examining each effect, and determining all the possible source locations that effect might be located in the 32-bit instruction. The template ids are grouped according to the source field (if any) that they use for the effect, and a multiplexer is constructed that chooses the appropriate field (or nop) for each group of ids. As shown in Figure 4.3, the template formats are selected so that the positions of fields overlap as much as possible, in order to minimize the number of inputs required for each multiplexer. In practice, no multiplexer had more than 5 inputs, The complexity of this stage is much lower than the execute stage, and easily fits within the same timing constraints.

When a PTB effect is active in the template expansion stage, the corresponding BEQ or BNE effect is expected to be in the fetch stage. The result of this BEQ/BNE effect is used to select the program counter to be used for the next fetch. Thus, at the end of the template expansion stage, the PTB field of the expanded instruction (combined with the prediction provided by the BPB) is used to select the target that will be used for the next fetch. This target is forwarded from the values that will be written to the registers at the end of the execute stage, and is saved in the

Table 4.2: Pipeline Configuration

L1 Miss Latency	16 cycles
L1-IC/DC	4-way assoc., 512 sets, 16 B line (32KB), LRU replacement
BPB	Bimodal, 32 entries, 2-bit state
BTB	2-way assoc., 32 sets, LRU replacement
Multiply Latency	10
Divide Latency	20
Clock	250 MHz

register labelled TGT. If a misprediction is discovered in the execute stage, later instructions are flushed, and fetch resumes from the correct program counter.

Since the LV register can be used as an input to the ALUs, the access to memory must be initiated in the template expansion stage, so that the data that will be written to the LV register will be ready before the instruction retires. This requires that the value to be written in the ADDR1 register at the end of the execute stage be forwarded to the end of the template expansion stage. Additionally, the data to be stored must be selected from the new register values that will be written at the end of the execute stage.

Similarly configured implementations of the Static Pipeline and the OpenRISC pipeline described in previous sections were synthesized using a low-power 65 nm technology by ST Microelectronics. The configurations for these processors are shown in Table 4.2. The static pipeline does not require a BTB, and so this entry does not apply in that case. The critical path for each design was through the L1 data cache tag check.

The multiplier and divider used for this study are respectively synthesized from components in the DesignWare library that are included with Synopsys Design Compiler. The multiplier used is the **dw_mul_seq** sequential multiplier, and the divider used is the **dw_div_seq** sequential divider. These components have latencies (in cycles) that are adjustible. The desired number of cycles is provided as a parameter when instantiating the component, and Design Compiler chooses an implementation that will complete in the given number of cycles. Choosing a larger cycle count allows the tool to create a component with lower cycle times, and vice versa. For each component,

Table 4.3: Pipeline Area Results

	OpenRISC	Static Pipeline
L1-IC	459406	459427
L1-DC	497640	496119
Pipeline	102828	52277
Control	4864	2970
Datapath	13625	17408
Functional Units	12411	13117
BPB	1152	1152
BTB	53117	—
Register File	17651	17626
Miscellaneous	8	4
Miscellaneous	3544	3197
Total Area	1063420	1011022

the cycle count was chosen to be the smallest number of cycles such that the component's cycle time was shorter than the L1 data cache cycle time.

The resulting area for each synthesized pipeline is shown in Table 4.3. The static pipeline has a smaller area primarily due to the lack of a BTB. Additionally, the static pipeline's control logic is slightly smaller, due to the simplicity of the pipeline and reduced instruction decoding logic. On the other hand, the datapath area is significantly larger due to the large number of multiplexers required to shuttle values between registers and functional units. The static pipeline also has an additional 32-bit adder not found in the OpenRISC, which is needed for the ADDR1 effect and accounts for the additional functional unit area. In the OpenRISC pipeline, the adder is shared for both add instructions and load/store address calculation.

4.3 Experimental Results

After synthesis, the netlists were simulated using the Mibench benchmark suite to gather switching statistics for the design. This information was fed into Synopsys Design Compiler to calculate power consumption for these benchmarks. Since support for floating-point operations adds significant complexity to processors, they were not used in this study. Thus, the benchmarks in the

Mibench suite that use floating point were not included. Additionally, the high-level simulations used in previous studies simulated a small fraction of instructions (approximately 1%) using the standard MIPS ISA. This MIPS code was used to simplify the implementation of the simulator, which is based on the SimpleScalar toolset, and is used exclusively for system calls. The simulator used in this study does not support executing both static pipeline instructions and MIPS instructions, since it realistically simulates the actual hardware. However, this was not a problem as the benchmarks have been modified to remove all uses of system calls.

Figure 4.5 shows the relative execution times in cycles of both the Static Pipeline and the OpenRISC on the integer subset of the Mibench benchmark suite. On average, the Static Pipeline performs nearly 7% better than the OpenRISC pipeline. Most of the improvement comes from the 5% reduction in IPC, from 0.77 for the OpenRISC, to .95 for the static pipeline. The static pipeline is able to achieve this higher IPC for several reasons. First, the static pipeline does not need to stall due to data hazards. Second, the static pipeline has fewer branch mispredictions because it can always provide the correct branch target without using a BTB. Third, when a branch is mispredicted, the penalty is lower due to a shorter pipeline. Additionally, the static pipeline requires 1% fewer instructions to complete the same work, as shown in Figure 4.6.

Figure 4.7 shows the power used by each pipeline. The power for each processor is broken down into the following components:

- Pipeline (control and datapath)
- L1 instruction cache (SRAMs and controller)
- L1 data cache (SRAMs and controller)
- Miscellaneous (cache arbiter, etc.)

Figure 4.8 breaks down the power used by each subcomponent of the pipelines, excluding caches. Pipeline power is broken down into the following subcomponents:

- **SP Register File**: static pipeline register file

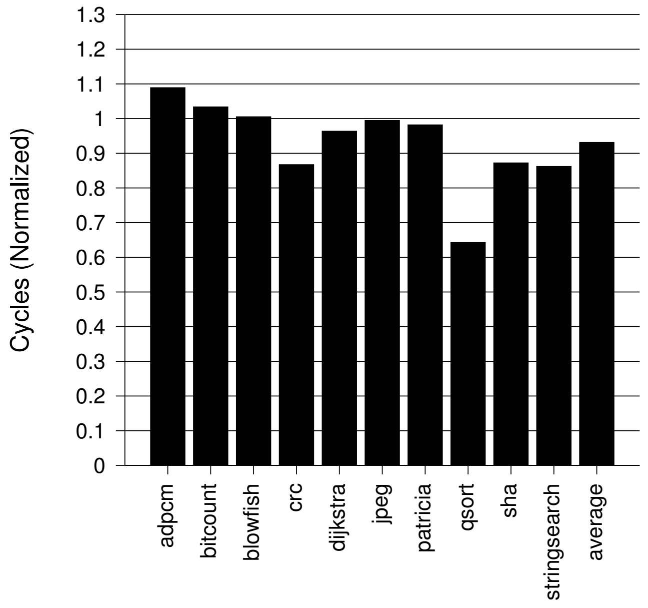


Figure 4.5: Static Pipeline Relative Cycles

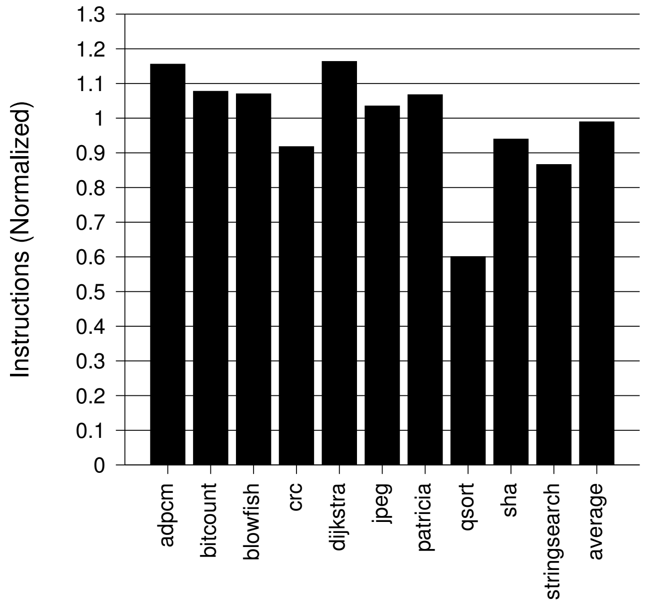


Figure 4.6: Static Pipeline Relative Instructions

- **SP ALU**: the adders, multiplier, divider, and shifter required to implement the static pipeline's FU1, FU2, and ADDR1 effects
- **SP Datapath**: static pipeline datapath, including internal registers
- **SP Control**: static pipeline control, including instruction expander
- **SP BPB**: static pipeline branch prediction buffer
- **OR1K Register File**: OpenRISC register file
- **OR1K ALU**: the OpenRISC adders, multiplier, divider, and shifter
- **OR1K Datapath**: OpenRISC datapath, including internal registers
- **OR1K Control**: OpenRISC control, including instruction expander
- **OR1K BTB**: OpenRISC branch target buffer
- **OR1K BPB**: OpenRISC branch prediction buffer

The Static Pipeline itself uses less than half the power of the OpenRISC. The vast majority of the static pipeline's power reduction comes from the removal of the BTB, which uses over half the power of the RISC pipeline. Additional small power reductions come from reductions in register file activity. However, the static pipeline's datapath power is somewhat higher, which is primarily due to the number of large multiplexers required. The ALU power of the static pipeline is also higher, due to the second adder required for the ADDR1 effect. The OpenRISC shares a single adder for both ALU operations and load/store address calculations.

Most significantly, the L1 instruction cache power usage is much higher for the static pipeline. This is explained by the higher IPC found in the static pipeline. The increase in instruction cache power does not completely undo the power saved by the static pipeline, which shows an average total core power reduction of nearly 2%.

Though power is reduced somewhat, the performance improvement is more significant, and thus the primary benefit of the static pipeline is actually performance, and not power reduction.

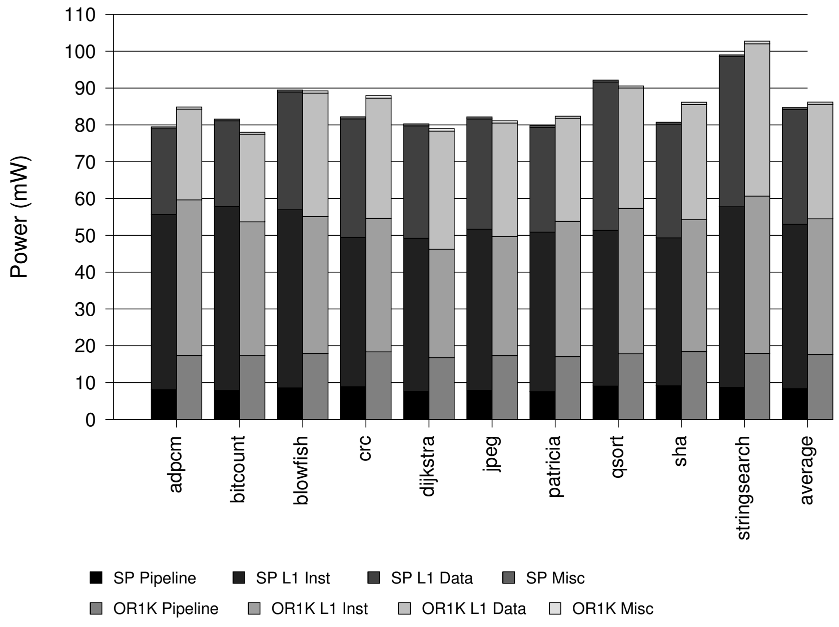


Figure 4.7: Static Pipeline Power

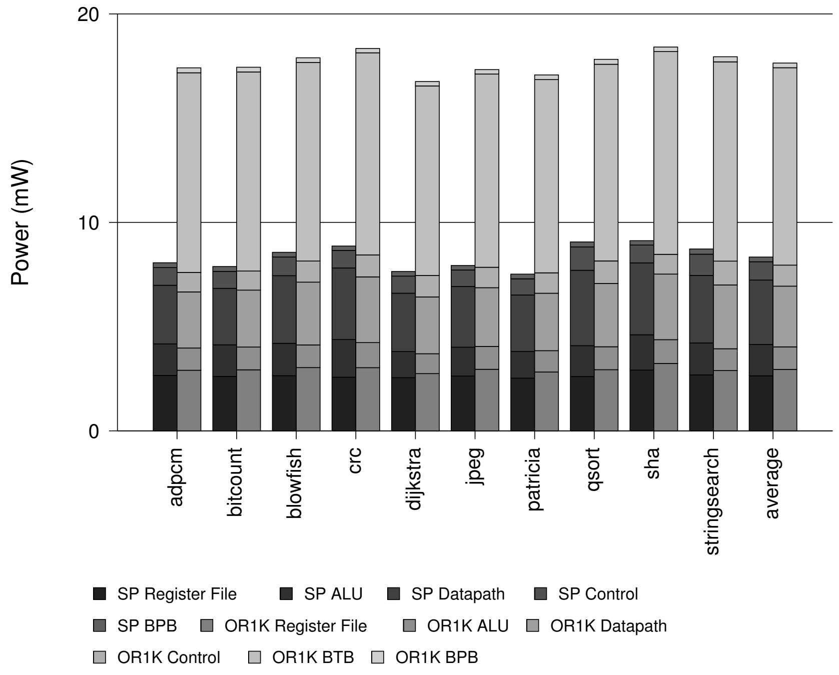


Figure 4.8: Pipeline-only Power

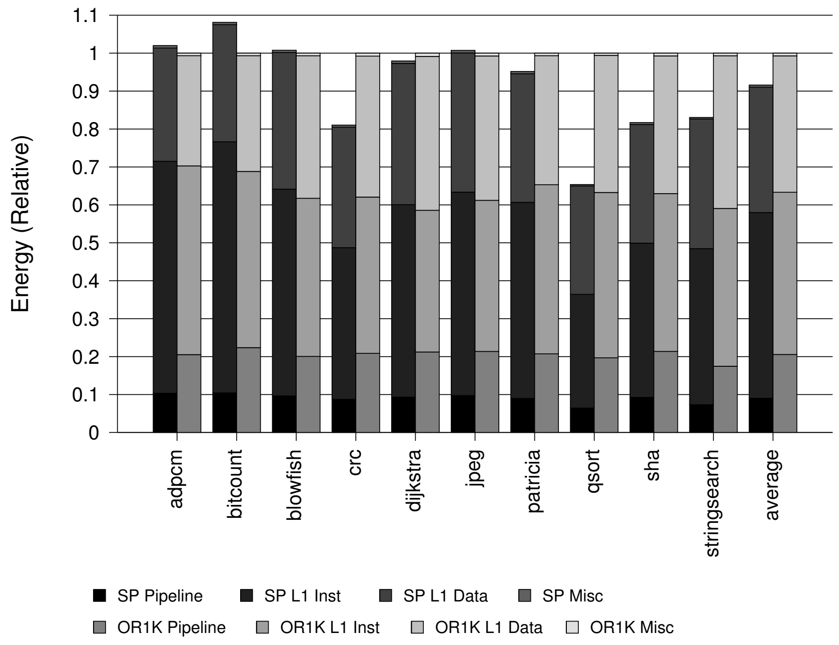


Figure 4.9: Static Pipeline Energy

However, improving performance while using the same power translates directly into energy savings. Figure 4.9 shows the energy used by each pipeline. The slight decrease in power usage, combined with the shorter execution time results in an overall reduction in energy usage of 8.5%. The combination of improved performance and reduced power results in even larger improvements to the energy*delay product and energy*delay^2, as shown in Figures 4.10 and 4.11. The energy*delay product is improved by over 13%, and the energy*delay^2 is improved by over 16%.

Earlier work assumed that the static pipeline would be able to reduce power and energy usage by reducing activity in the register file, branch prediction structures, internal (pipeline) registers, and so on. These assumptions appear to be confirmed in the resulting pipeline power usage rates. Additionally, it was assumed the compiler could can do a better job of scheduling pipeline actions than is possible in the tightly constrained logic found in a microprocessor. This assumption is confirmed in the reduced cycle counts found in the static pipeline. However, the higher IPC found in the static pipeline results in higher cache power, reducing the power saved by the pipeline itself. The additional instruction fetch energy was not accounted for in earlier studies, and thus most of

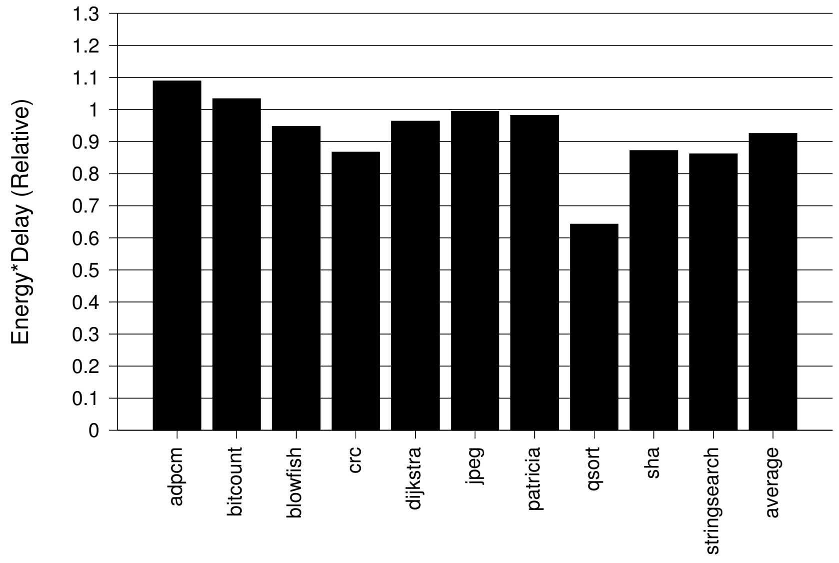


Figure 4.10: Static Pipeline Energy*Delay

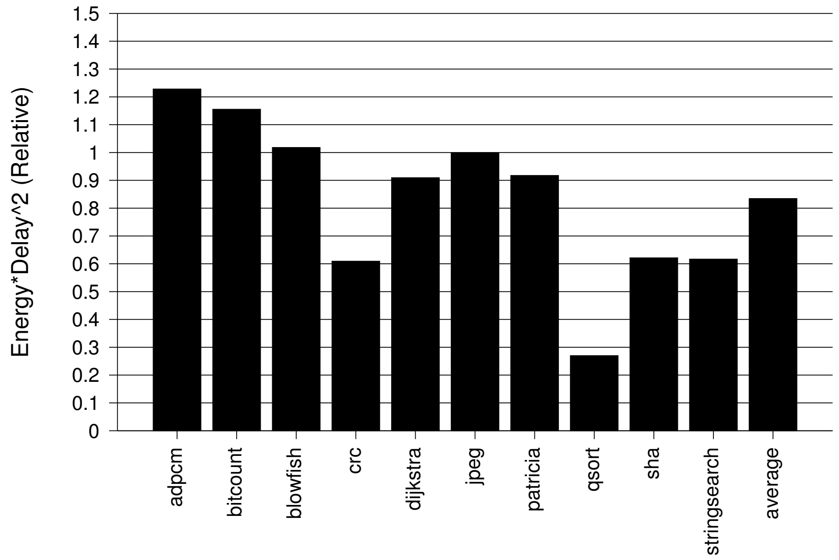


Figure 4.11: Static Pipeline Energy*Delay^2

the power usage was not reduced, but instead moved to a different component in the processor.

As mentioned, the removal of the BTB provides a significant reduction in pipeline power, and deserves some discussion. RISC architectures frequently use a branch delay slot, which gives the pipeline an additional cycle to resolve branches, in which case a BTB is not necessary. However, using a delay slot results in a performance decrease. It is often the case that the delay slot cannot be filled with a suitable instruction, such as when the branch condition is dependent on all the instructions in the basic block. Additionally, there may be data hazards on the values used to resolve the branch, especially since the branch condition must be resolved at such an early stage in the pipeline. This means that additional forwarding paths will be required, and that the pipeline must sometimes insert bubbles when those values cannot be forwarded. The static pipeline does not have these problems, as it uses a BPB to speculate on the condition, and the BPB is typically quite accurate and uses comparatively little power.

The high-level studies of the static pipeline included floating-point benchmarks, for which the static pipeline was able to provide better power savings than are seen for the integer benchmarks. Floating benchmarks tend to be dominated by small loops that execute for many iterations, and the static pipeline performs quite well on this type of code. The performance benefits of static pipelining for small loops are evident in the **qsort**, **crc**, **sha**, and **stringsearch** benchmarks, whose runtimes are dominated by small loops.

CHAPTER 5

RELATED WORK

Other architectural features related to the TH-IC and static pipelining are presented in this section.

5.1 Loop Buffers

Loop buffers are small structures that have been proposed for fetching innermost loops using less energy. Their purpose and is very similar to the TH-IC: to reduce the number of L1-IC and I-TLB accesses by retrieving the instructions from a much smaller, lower power structure. Not only do loop buffers avoid L1-IC accesses, but they also do not require I-TLB accesses while executing from the loop buffer. Loop buffers are automatically exploited when a short backward direct transfer of control is encountered, indicating that looping behavior is likely [19]. In order to reduce the rate of false positives, many loop buffer designs will not become active until one or more iterations have occurred.

The simplest of loop buffers can only buffer loops with no (or minimal) control flow. More advanced designs, such as the loop buffer found in the Cortex A15 [18], are capable of handling one or more forward branches, in addition to the backward branch at the end of the loop. This allows simple if/else sequences to be handled, while still fetching from the loop buffer.

We have simulated our baseline models augmented with an “ideal” loop buffer, with sizes equivalent to the sizes of the TH-IC studied. This loop buffer is ideal, in that it can buffer any innermost loop with simple internal control flow. The simulated loop buffer can handle an arbitrary number of forward branch instructions. It can also handle an arbitrary number of backward branch instructions, provided that these backward branches always target the first instruction of the loop. On recovery from a branch misprediction, the loop buffer resumes the most recently buffered loop, if any, unless the misspeculated path results in a new loop being detected. Other backward branches, and all indirect jumps and returns cause the loop buffer to become idle. If an innermost loop exceeds the size of the loop buffer, only the initial instructions of the loop will be buffered, and later instructions will be accessed from the L1-IC. Loops will only be buffered if they meet the above conditions, and after two iterations of the loop have passed.

In nearly all cases the simulated loop buffer hit rate was significantly lower than the TH-IC hit rate (for the same number of instructions stored), and in no case was the loop buffer hit rate over 50%. Figure 5.1 shows the fraction of instruction fetches that occur from the loop buffer (hit rate), for different loop buffer sizes on a single fetch width core. In no case is the hit rate greater than 50%, while the TH-IC easily manages to exceed 75% of accesses from the TH-IC. (Hit rates for different fetch widths were nearly identical.) The mediocre hit rate achieved by the loop buffer strategy is directly tied to their limited flexibility with regard to control flow. The TH-IC does not have this problem, and can effectively handle loops containing arbitrary conditional and non-conditional control flow. Nested loops are easily handled by the TH-IC. The TH-IC can also recover quickly from irregular control flow, such as indirect jumps, which would cause the loop buffer to go idle until the next suitable innermost loop is detected.

As a result of the low hit rate, loop buffer does not reduce fetch energy nearly as well as the TH-IC. Figures 5.2-5.4 show the simulated energy usage of the baseline pipelines, augmented with loop buffers of various sizes, under 10% static leakage conditions. In these figures, the notation "$X/\mathrm{L}Y$" refers to a configuration with an X-byte IC line size, and a loop buffer with storage for Y instructions. At best, the ideal loop buffer saves 20-25% of the energy used for instruction fetch, while the TH-IC achieves savings in excess of 40%. In addition, the loop buffer simulated here has fewer restrictions than any loop buffer found in an actual processor, and so the hit rates achieved in reality would likely be lower.

5.2 Low Power Instruction Fetch

Zero overhead loop buffers (ZOLBs) are small structures sometimes used in embedded processors [9]. Unlike the loop buffer described above, the ZOLB is explicity initialized by the compiler

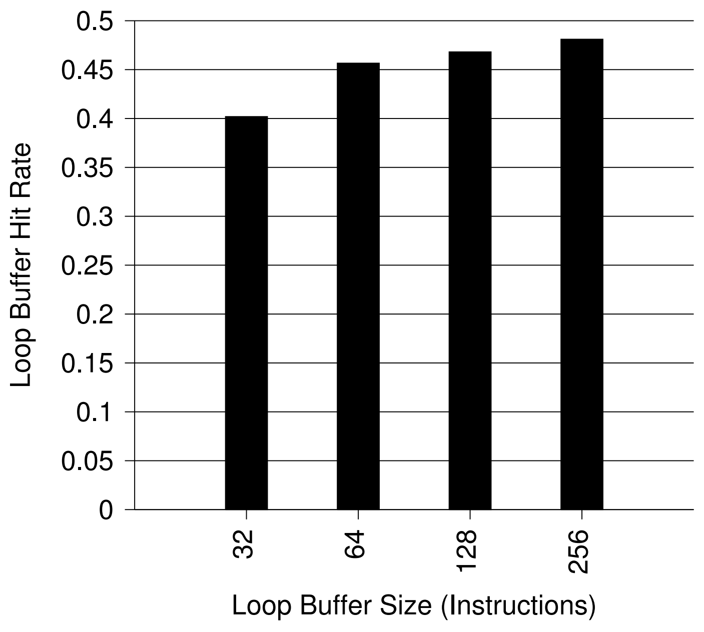


Figure 5.1: Loop Buffer Hit Rate

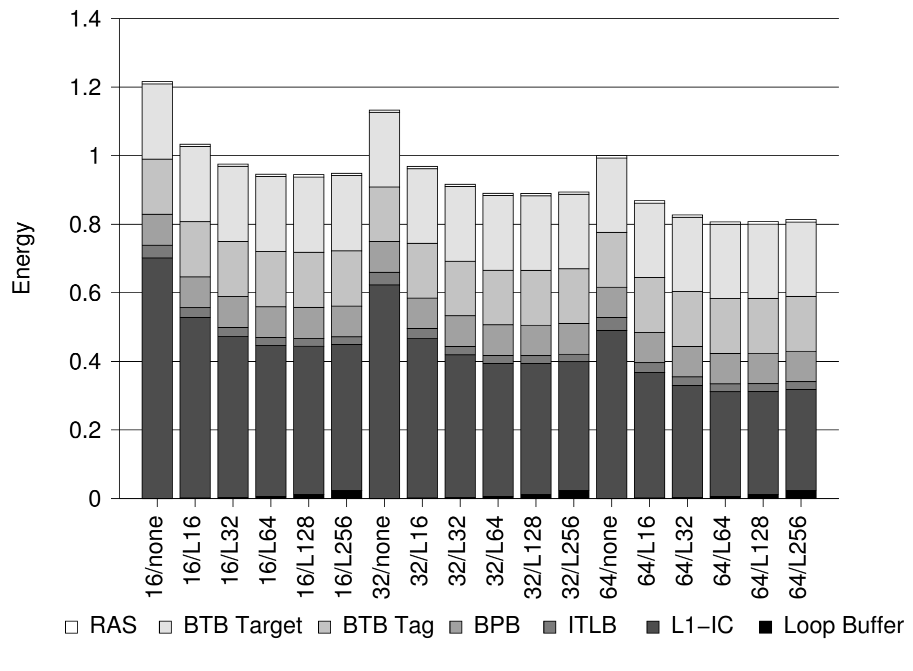


Figure 5.2: Loop Buffer Fetch Energy, 1-wide Fetch

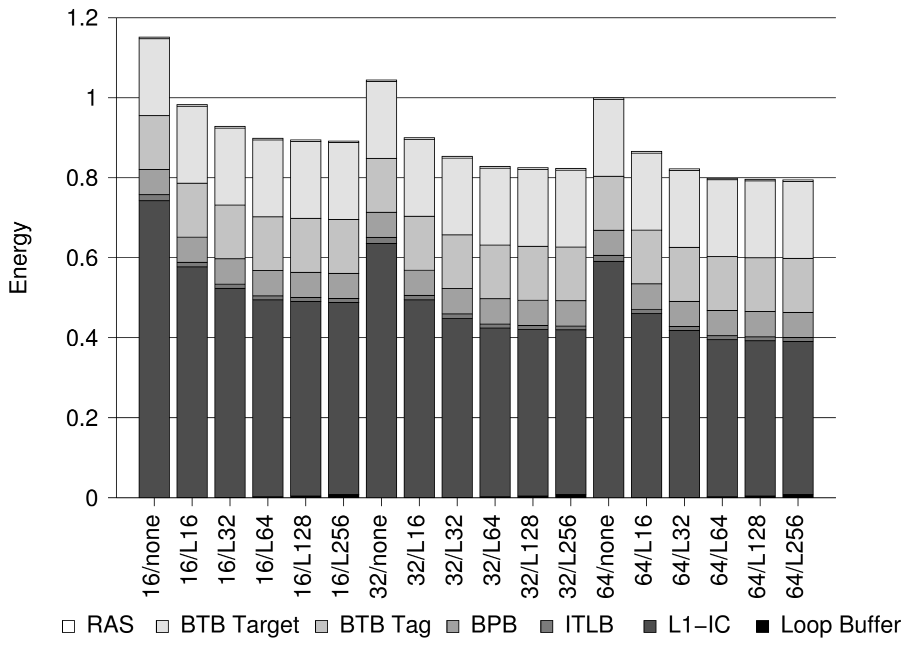


Figure 5.3: Loop Buffer Fetch Energy, 2-wide Fetch

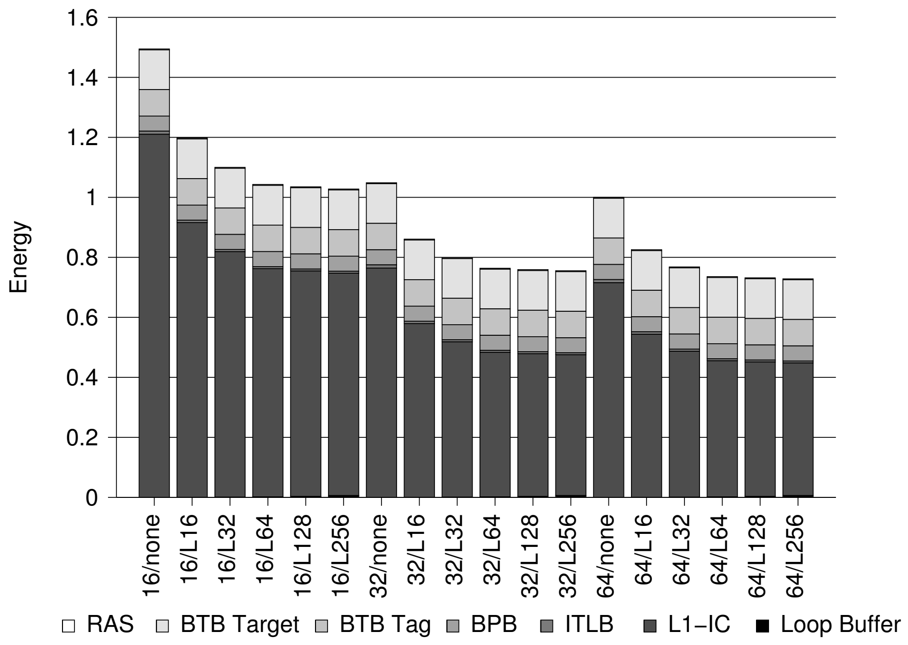


Figure 5.4: Loop Buffer Fetch Energy, 4-wide Fetch

using instructions introduced for that purpose[26, 27]. The ZOLB benefits include reducing loop overhead due to using an internal counter and avoiding L1-IC, I-TLB, and BPB/BTB accesses after the first iteration of the loop. However, a ZOLB also provides limited benefit as the loops it can exploit must have a known number of iterations, must have no conditional control flow except for the loop branch, and must be small enough to fit into the ZOLB. In contrast to both of these approaches, our techniques can disable access to the L1-IC, I-TLB, and BPB/BTB much more frequently.

The trace cache [25, 24] was developed to widen the instruction fetch bottleneck by allowing instruction fetch accesses to traverse transfers of control. It achieves this by concatenating multiple separate basic blocks into a single cache line, which is indexed by the initial PC combined with the taken/not-taken states of each branch in the sequence of blocks. Since the cache is indexed using the branch pattern of the sequence, different branching patterns will lead to duplication of basic blocks within the trace cache. The size of the trace cache is typically fairly close to the size of the L1 cache, while the size of the TH-IC is much smaller. These factors combine to make trace caches less suitable for low-power, embedded architectures.

Kadayif et al. studied not accessing the I-TLB unless the virtual page changes [16]. One technique they evaluated was to examine the updated PC (sequential accesses, BTB lookups for branches, etc.) and compare the virtual page number with the current virtual page number. This approach works well as long as the comparison being performed does not affect the critical path for the cycle time as the comparison occurs after a new PC value is obtained, which is typically near the end of the cycle. In contrast the techniques for disabling the I-TLB in this dissertation should not put extra logic on the critical path. For instance, we determine if the next sequential fetch will cross a page boundary by checking at the beginning of the cycle if the address of the current instruction is the last instruction in the page and is only required to perform this check when the last line in the TH-IC is the last line in the page. We also detect when the target of a ToC instruction is to the same page and sets an SP bit that is used to disable the I-TLB access in the next cycle if the ToC is taken. Kadayif et al. also suggested that the format of ToC instructions could be changed to indicate whether or not the target is to the same page. We capture most of this benefit while not requiring any ISA changes.

There have also been proposed alternative BPB and BTB designs to reduce energy usage. A leakage power reduction technique has been proposed to make BPB and BTB entries drowsy if they are not accessed after a specified number of cycles [15]. The use of counters will offset some of the energy benefits. Another BTB design proposed using fewer tag bits to save power at the cost of some false positive tag matches [10]. We believe that this paper's proposals are complementary to both of these techniques, and could be advantageous in combination, as the number of accesses are reduced, and so more targets are available for activation of drowsy mode. Yang et al. proposed a small branch identification unit and avoided accessing the BTB until the next branch is encountered. But this approach requires changing executables to store branch distance information and also requires the use of a counter to detect the next branch [29].

Parikh et al. proposed disabling access to the BTB and BPB to save energy by using a *prediction probe detector* (PPD) that contains two bits for each instruction in the L1-IC [23]. These two bits indicate whether or not the BPB and BTB should be accessed and these bits are assigned values while an L1-IC cache line is being filled. These authors proposed that the PPD be checked before going to the BPB or BTB, which they recognized may sometimes be on the critical timing path. In addition, the predecode bits used in this paper are used to set NSNB bits when a line is loaded from the L1-IC into the TH-IC. In contrast to the PPD approach, the techniques presented in this paper do not affect the critical timing path.

Calder et al. proposed making all branches use page offsets instead of PC offsets to allow fast target calculation [7]. These authors proposed to use indirect jumps through a register for branches across page boundaries. They also used predecode bits to indicate whether or not an instruction is a branch. The disadvantages of this approach are that it requires an ISA change and the page offset extraction from the L1-IC may be on the critical timing path.

Hines et. al. [13] introduces NSNB and NTNB bits in order to disable branch prediction structures for non-ToCs. That work does not discuss a method for using these bits on multiple-fetch processors. Additionally, it proposes waiting until instructions are fetched and decoded to detect whether or not it is a transfer of control, and requires that this detection is completed before the end of the fetch stage. By using predecoding as described in our research, these bits can be used before an instruction is even accessed, dramatically increasing their effectiveness. Hines et. al. does not take advantage of the other techniques listed in this dissertation for disabling the I-TLB and BTB tag arrays. We were able to achieve much greater power reduction with these additional techniques than was found in that paper.

Finally, Nicolaescu et. al. [21] describes a technique for reducing data cache access energy that may be adapted to instruction fetch. Data access instructions require that the address is generated by adding an immediate offset to a base value retrieved from a register. The offset is typically small, and adding it to the base often does not produce carry-outs from many of the most significant and least significant bits. This property can be exploited to use a fast adder that does not propagate these carry-outs, and instead only checks them to detect when the result will be incorrect. The "fast" address is then used to access a "way cache unit", a small, fully-associative structure that memoizes the way that was last used to access the particular address. The way cache unit is a small enough that it can be accessed in the same cycle as and immediately after the fast address generation. If the way cache hits, the retrieved way is used to access the L1-DC, without requiring access to the tag array or D-TLB (Data Translation Lookaside Buffer).

5.3 Transport Triggered Architectures

The Static Pipeline has been compared to other "exposed datapath" architectures. One of these architectures is the "Transport Triggered Architecture" (TTA).[8] In a transport triggered architecture, the only instruction is a "move" instruction, that transports data from one register to another. Some registers, however, are attached to the inputs of functional units, and can "trigger" an operation when data is moved into them. Typically, multiple moves can be completed in parallel each cycle, and some TTA implementations may allow moves to be predicated by some condition. Functional units have zero or more operand registers, and a trigger register, which also recieves an operand but initiates the operation.

A standard RISC-like instruction requires multiple TTA "instructions" to complete. For example, consider a simple instruction from a RISC ISA, such as:

$\mathsf{add}\quad r1, r2, r3$

On a transport triggered architecture, this instruction would instead be implemented as:

$r2 \rightarrow \mathsf{add.operand};\quad r3 \rightarrow \mathsf{add.trigger}$

$\mathsf{add.result} \rightarrow r1$

The first instruction moves the first operand into the operand register for the functional unit responsible for addition. The next moves the second operand into the trigger register for the functional unit. The move into the trigger register causes the add to be initiated. The third instruction reads the sum of the operands out of the functional unit's result register.

Like the Static Pipeline, forwarding of values between operations can occur without writing intermediate values to the register file. For example, the RISC-like sequence

$\mathsf{add}\quad r1,\quad r2,\quad r3$

$\mathsf{add}\quad r4,\quad r1,\quad r5$

can be achieved with TTA code sequence

$r2 \rightarrow \mathsf{add.operand};\qquad\qquad r3 \rightarrow \mathsf{add.trigger}$

$\mathsf{add.result} \rightarrow \mathsf{add.operand};\quad r5 \rightarrow \mathsf{add.trigger}$

$\mathsf{add.result} \rightarrow r4$

Here, the write to $r1$ by the first RISC instruction is avoided by forwarding the result of the first add directly back to the operand port of the adder without going through the register file. The corresponding Static Pipeline effects would be:

$$\begin{array}{ll} RS1 = r2; & RS2 = r3 \\ OPER1 = RS1 + RS2; & RS2 = r5 \\ OPER1 = OPER1 + RS2 & \\ r4 = OPER1 & \end{array}$$

Again, the write of the intermediate value to $r1$ is avoided.

Unlike the static pipeline, a TTA requires a large transport network to shuffle values between the register file and the result and operand registers of the functional units. The static pipeline *does* require something similar to a transport network, but its size is limited by restricting value movement to a set of paths similar to what might be found in a simple RISC processor. This is possible because the Static Pipeline compiler produces code by statically expanding each RISC instruction into a set of effects that mimics what is performed dynamically by a 5-stage pipeline. Similar techniques could be used to generate TTA code, but the primary goal of TTA is high performance (as opposed to low power) which is facilitated by the ability to transport values arbitrarily.

CHAPTER 6

FUTURE WORK

This chapter describes future work for the TH-IC and Static Pipeline.

6.1 Tagless-Hit Instruction Cache

The study of the TH-IC could be improved by examining architectures with wider pipelines and fetch windows. A more thorough implementation including place and route and clock tree insertion would allow the capacitance of the final circuit to be extracted, resulting in a more accurate power usage estimate.

Finally, the proposed extensions that were only simulated at a high level could be implemented as was done for the TH-IC.

6.2 Static Pipeline

The most fruitful method for reducing the static pipeline's power usage would likely be to reduce its dynamic instruction counts, which are high when compared to a traditional RISC architecture. This could be achieved by finding a more compact encoding, and by further improvements to the code generated by the compiler. It may involve rearchitecting the pipeline as well. For example, tradeoffs that improve code density by pushing some work back down to the processor may improve total power usage by reducing the activity of the power-hungry instruction cache at the expense of higher pipeline power.

CHAPTER 7

CONCLUSION

This dissertation presents a detailed analysis of a pair of techniques for reducing power and energy in microprocessors. The first of these techniques, the Tagless-Hit Instruction Cache, is quite successful at reducing instruction fetch power and energy usage. Caches in general usage a substantial fraction of total core power, and the instruction cache is particularly power hungry, since it is activated nearly every cycle in order to provide instructions to the pipeline. In comparison, the data cache does not use as much power, since loads and stores occur in only a fraction of all instructions. By fulfilling instruction fetch requests from a small tagless array, the TH-IC is able to reduce instruction fetch energy by nearly 20%, and total core energy by over 7%. For the baseline processor used in this dissertation, timing was not impacted, and performance was reduced by less than 2%. For situations where power usage is a concern, this slight performance reduction may not be an issue, and will be outweighed by the benefits in power and energy usage. More advanced modifications to the TH-IC have been shown in high-level simulation to provide even greater reductions in energy. However the timing impact of these modifications have not been evaluated, and they have not been simulated at the netlist level to provide more confidence in their energy reductions.

For the second technique, static pipelining, this dissertation was able to show significant energy savings. These energy savings come primarily from improvements in performance while using similar power to traditional RISC. The static pipeline itself uses significantly less power than a traditional RISC pipeline (up to 50%), however caches are the dominant source of power dissipation. This study found that static pipelining achieves a higher IPC, resulting in performance is that is better than that of a traditional in-order 5-stage pipeline. This higher IPC results in higher instruction cache power, thus negating some of the power reductions of the static pipeline. However, the performance is sufficient to result in substantial energy reductions.

Further improvements to performance and energy usage may be possible, over the savings found thus far. The static pipeline model produced for this dissertation did not include floating point support. High level simulation found that floating benchmarks use significantly lower power when executed on the static pipeline. It is likely that these finding will carry over to the real hardware, as was the case for the integer-only benchmarks used in this study. Additionally, due to the exposed nature of the static pipeline, future research may discover optimizations that apply to the static pipeline and not to a traditional RISC pipeline.

The results of both studies, when considered together, show that optimization effort should be directed toward eliminating large structures such as caches, and reducing their activity. In comparison, the logic required to implement most ISA-specific features such as instruction decoding, result bypassing, register file accesses, ALU operations, and so on is quite small. On-core caches use considerable power, especially in the case of the instruction cache, which must be accessed for every single instruction that is fetched. The BTB, which is also accessed at every instruction fetch, can consume a substantial portion of pipeline power. The power needed to implement the simple abstractions provided by traditional ISAs is in fact not significant.

Techniques that reduce cache access rates are likely to be provide tremendous benefits in terms of reduced power. For example, the Thumb instruction set supported by ARM processors provides improved code density, thus allowing the same program to be executed with fewer instruction cache accesses. Though instruction decode complexity is increased, it allows twice as many instructions to be fetched per cycle without increasing cache activity. However, there are some limitations imposed by the smaller instruction size, such as fewer accessible registers and fewer available operations, and thus performance may be reduced.

These studies have also shown the importance of detailed and realistic simulation when attempting to estimate power usage of new designs. Earlier studies found reduced power and energy usage, and these reductions were confirmed in lower level studies, but were not as significant. When limiting power and energy usage is the primary design goal, detailed simulations should be performed at the earliest feasible stage of the design process.

BIBLIOGRAPHY

[1] CARPE, November 2014.

[2] ARM Holdings. Arm information center. Website, 2013.

[3] Todd Austin, Eric Larson, and Dan Ernst. SimpleScalar: An infrastructure for computer system modeling. *IEEE Computer*, 35:59–67, February 2002.

[4] A. Bardizbanyan, M. Sjalander, D. Whalley, and P. Larsson-Edefors. Designing a practical data filter cache to improve both energy efficiency and performance. *Transactions on Architecture and Code Optimization*, 10(4), December 2013.

[5] David Brooks, Vivek Tiwari, and Margaret Martonosi. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th annual International Symposium on Computer Architecture*, pages 83–94, New York, NY, USA, 2000. ACM Press.

[6] Doug Burger and Todd M. Austin. The SimpleScalar tool set, version 2.0. Technical Report 1342, University of Wisconsin - Madison, Computer Science Dept., June 1997.

[7] Brad Calder and Dirk Grunwald. Fast and accurate instruction fetch and branch prediction. In *International Symposium on Computer Architecture*, pages 2–11, June 1994.

[8] H. Corporaal. Design of transport triggered architectures. In *VLSI, 1994. Design Automation of High Performance VLSI Systems. GLSV '94, Proceedings., Fourth Great Lakes Symposium on*, pages 130–135, 1994.

[9] J. Eyre and J. Bier. DSP processors hit the mainstream. *IEEE Computer*, 31(8):51–59, August 1998.

[10] Barry Fagin and Kathryn Russell. Partial resolution in branch target buffers. In *ACM/IEEE International Symposium on Microarchitecture*, pages 193–198, 1995.

[11] Global Foundries. Global foundries 32/28nm hkmg platforms. Website, 2011.

[12] Matthew R. Guthaus, Jeffrey S. Ringenberg, Dan Ernst, Todd M. Austin, Trevor Mudge, and Richard B. Brown. MiBench: A free, commercially representative embedded benchmark suite. *IEEE 4th Annual Workshop on Workload Characterization*, December 2001.

[13] Stephen Hines, Yuvall Peress, Peter Gavin, David Whalley, and Gary Tyson. Guaranteeing instruction fetch behavior with a lookahead instruction fetch engine. In *Proceedings of the ACM Conference on Languages, Compilers, and Tools for Embedded Systems*, pages 119–128. ACM, June 2009.

[14] Stephen Hines, David Whalley, and Gary Tyson. Guaranteeing hits to improve the efficiency of a small instruction cache. In *ACM/IEEE International Symposium on Microarchitecture*, pages 433–444. IEEE Computer Society, December 2007.

[15] Z. Hu, P. Juang, K. Skadron, D. Clark, and M. Martonosi. Applying decay strategies to branch predictors for leakage energy savings. In *Proceedings of the International Conference on Computer Design*, pages 442–445, September 2002.

[16] I. Kadayif, A. Sivasubramaniam, M. Kandemir, G. Kandiraju, and G. Chen. Generating physical addresses directly for saving instruction tlb energy. In *Proceedings of the 40th annual ACM/IEEE International Symposium on Microarchitecture*, pages 433–444. IEEE Computer Society, December 2007.

[17] Johnson Kin, Munish Gupta, and William H. Mangione-Smith. Filtering memory references to increase energy efficiency. *IEEE Transactions on Computers*, 49(1):1–15, 2000.

[18] Travis Lanier. Exploring the design of the cortex-a15 processor. Presentation slides, 2011.

[19] L. Lee, B. Moyer, and J. Arends. Instruction fetch energy reduction using loop caches for embedded applications with small tight loops. In *Proceedings of the International Symposium on Low Power Electronics and Design*, pages 267–269, 1999.

[20] Linux Kernel Developers. Linux KConfig, November 2014.

[21] D. Nicolaescu, B. Salamat, A. Veidenbaum, and M. Valero. Fast speculative address generation and way caching for reducing l1 data cache energy. In *Computer Design, 2006. ICCD 2006. International Conference on*, pages 101–107, 2006.

[22] OpenRISC Maintainers. OpenRISC, November 2014.

[23] D. Parikh, K. Skadron, Y. Zhang, M. Barcella, and M. Stan. Power issues related to branch prediction. In *Proceedings of the International Symposium on High Performance Computer Architecture*, pages 233–244, February 2002.

[24] Eric Rotenberg. A trace cache microarchitecture and evaluation. *IEEE Transactions on Computers*, 48:111–120, 1999.

[25] Eric Rotenberg, Steve Bennett, James E. Smith, and Eric Rotenberg. Trace cache: a low latency approach to high bandwidth instruction fetching. In *In Proceedings of the 29th International Symposium on Microarchitecture*, pages 24–34, 1996.

[26] Gang-Ryung Uh, Yuhong Wang, David Whalley, Sanjay Jinturkar, Chris Burns, and Vincent Cao. Effective exploitation of a zero overhead loop buffer. In *Proceedings of the ACM SIGPLAN Workshop on Language, Compilers, and Tools for Embedded Systems*, pages 10–19, May 1999.

[27] Gang-Ryung Uh, Yuhong Wang, David Whalley, Sanjay Jinturkar, Chris Burns, and Vincent Cao. Techniques for effectively exploiting a zero overhead loop buffer. In *the Proceedings of the International Conference on Compiler Construction*, pages 157–172, March 2000.

[28] S. Wilton and N. Jouppi. Cacti: An enhanced cache access and cycle time model. *IEEE Journal of Solid State Circuits*, 31(5):677–688, May 1996.

[29] Chengmo Yang and Alex Orailoglu. Power efficient branch prediction through early identification of branch addresses. In *International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, pages 169–178, October 2006.

BIOGRAPHICAL SKETCH

Peter Gavin was born January 4, 1980, in Bakersfield, California. He received Bachelor of Science degrees in Applied Mathematics and Computer Science from Florida State University in 2007. He receieved a Master of Science degree in 2010, also from Florida State University. He completed his Doctor of Philosophy degree in Computer Science at Florida State University in December, 2014, under the advisement of David Whalley and Gary Tyson.