An Evaluation of Deeply Decoupled Cores

Eren Kursun KURSUN@CS.UCLA.EDU
University of California Los Angeles
Computer Science Department
Los Angeles, CA 90095 USA

Anahita Shayesteh ANAHITA@CS.UCLA.EDU
University of California Los Angeles
Computer Science Department
Los Angeles, CA 90095 USA

Suleyman Sair SSAIR@NCSU.EDU
North Carolina State University
Department of Electrical and Computer Engineering
Raleigh, NC 27695-7256 USA

Tim Sherwood SHERWOOD@CS.UCSB.EDU
University of California Santa Barbara
Department of Computer Science
Santa Barbara, CA 93110 USA

Glenn Reinman REINMAN@CS.UCLA.EDU
University of California Los Angeles
Computer Science Department
Los Angeles, CA 90095 USA

Abstract

The trend towards larger structures and aggressive clock frequencies has been a fundamental driving force for modern microprocessor design. While one approach is to deeply pipeline any high delay structure, dependencies and critical loops have made it increasingly difficult to speed execution through extensive pipelining. One alternative is to remove larger structures from the critical path. We explore the ramifications of stripping all but the most necessary functionality out of the processing core, leaving only a tiny μ-core. Although past studies have shown the possibility to build decoupled structures for some individual helper structures, the impact of streamlining *all* of these structures at the same time has not been explored. Along with describing the challenges in decoupling the helper engines, we focus on the performance, power consumption and thermal behavior of the μ-core architecture. We use a detailed performance, power and thermal modeling in our analysis. Our results indicate that the μ-core provides a 20% reduction in power over a conventional monolithic core, while providing comparable performance (1% improvement on average). By dynamically configuring the helper engines to different application phases, an additional 13% power savings can be attained with only an average 3% degradation in performance. Our experimental analysis also show that the microcore architecture has favorable thermal behavior, with 86% fewer thermally-critical cycles compared to a monolithic core.

1. Introduction

Despite increases in the transistor count available for designing future generation processors, emerging technology trends including poor wire latency scaling, increased power density, and reduced transistor reliability threaten to limit processor performance. Future scaling trends challenge microarchitects in improving both clock rate and IPC [1].

Prior work aimed at improving single-threaded IPC involves the discovery and exploitation of instruction level parallelism (ILP). However, wire latency has been projected [1] to impact large pipeline structures more than smaller structures, increasing the latency of critical processor loops. This will impact the size of the branch prediction structures that help hide the overall pipeline latency, the cache structures that hide instruction and data memory latency, the physical register file that supports a large instruction window to provide more ILP, and the other auxiliary speculative structures that further boost ILP or hide memory latency. One approach to providing aggressive cycle times in the face of increasing latency is to deeply pipeline all aspects of the processor, from the branch predictor to the instruction wakeup logic to the cache memories.

While deep pipelining has been effective at increasing operating frequency, one of the reasons that performance lags behind that of the trends set by frequency is the increased latency to critical processor loops [2] from deeply pipelined structures. Furthermore, there is extra latency from routing complexity due to large structures [3]. Larger structures also contribute to the growing power density and thermal problems facing modern processor design through greater power dissipation and longer clock wires [4].

One alternative approach is to *decouple* or *factor out* large structures from critical pipeline loops. Some structures can be broken in two: a small structure that can hold enough state to provide low-latency performance, and a larger second level structure that can hold sufficient capacity to keep performance high. Other structures can be completely factored from the processor core and have inherent latency tolerance that allows them to avoid becoming critical loops in the processor. Such factoring not only helps to reduce critical loop latency, but can also simplify routing on the critical path and help reduce power from critical path structures. Throughout the paper, *factoring* will be used as a general term to describe both of these optimizations.

In this paper, we combine prior techniques in factoring into a cohesive framework and extend this paradigm to more of the processor core. In [5], the structures that are factored from the main processor pipeline are called *helper engines*. We maintain this naming convention, and refer to the part that remains after extensive factoring as the *μ-core*. A conventional architecture featuring larger structures on the critical path is referred to as a *monolithic* architecture.

While there may be a performance penalty in dividing the processor into a μ-core and helper engines, as measured by instructions committed per cycle, there are many advantages that outweigh these penalties. The primary advantage of such a technique is that the performance critical parts are factored from the larger, more latency tolerant structures. This allows us to target each of these two different domains with different circuit, architectural, or application level techniques. For example, we can target the μ-core with very aggressive, high-power,large transistors to achieve the best cycle time possible. On the other hand, we can use slower, less power-hungry, and more dense transistors for the helper engines. As a result, the μ-core architecture shows favorable power and thermal behavior.

Power consumption has become a critical design constraint for microprocessors in recent years [6]. As the number of transistors on the next generation microprocessors nears one billion, ever shrinking feature sizes and exponential rises in cooling costs will further complicate this problem. We believe that our μ-core architecture can provide significant power reduction opportunities, not only as the promising initial results of this study illustrate, but also by the flexibility the factored architecture paradigm creates for other more sophisticated power optimization techniques.

Yet another effect of the technology trends is processor heating. Thermal issues threaten critical aspects of microprocessor design, such as reliability, proper functionality, performance and cost. To make matters worse, the cost of cooling increases at an exponential rate for increased temperatures [7].

To tackle the thermal issues, dynamic thermal management (DTM) has become an integral part of microprocessor designs [8, 9, 10, 11, 12]. However, it is important to note that all DTM techniques cause performance degradation to some degree, as a result of reducing the clock frequency, decreasing the supply voltage or in some cases temporarily shutting down the entire chip. Thermally efficient architectures with less overall heating are extremely desirable, as they minimize the overhead due to DTM.

Our results reveal that the μ-core architecture provides favorable power and thermal behavior compared to a conventional monolithic architecture. As the power dissipation and on-chip temperatures continue to rise for the next generation processor designs, inherent power and thermal efficiency of architectures are expected to become even more important.

Prior reseach [13, 14, 15, 16] has explored factoring or decoupling certain parts of the processor pipeline. Our contributions over prior work include:

- An extension of the factoring paradigm to the entire processor pipeline: the μ-core architecture
- Analysis of which structures are suitable for factoring - and the individual challenges in factoring these structures
- An exploration of how far different structures can be factored
- A detailed power and performance modeling to analyze the benefits of the factored μ-core over an aggressively pipelined conventional core
- An investigation of helper engine tuning in order to achieve the best performance with minimum number of helpers and minimum power dissipation
- An evaluation of thermal benefits of the μ-core architecture over a conventional monolithic architecture

The rest of this paper is organized as follows. In Section 2, prior work on the helper engine paradigm is discussed. Section 3 describes our factored μ-core architecture. Simulation methodology and benchmark descriptions can be found in Section 4. Section 5 presents performance and power results and we conclude in Section 6.

2. Related Work

In this section we discuss the relationship between our design and the most relevant prior work. In Section 3, we will detail the prior work for each individual helper engine when discussing the implementation of that helper engine.

In [5], Smith proposes a processor implementation that consists of several distributed functional units, each fairly simple and with a very high frequency clock. These units communicate via point–to-point interconnections that have short transmission delays. He then describes how surrounding this simple core pipeline with *helper engines* that perform speculative tasks off the critical path results in enhanced overall performance. Since the helper engines are off the critical path, they can use slower transistors to reduce static power consumption. This is also the motivation behind our factored design, where the speculative structures are shrunk to a bare minimum size to support near ILP but they are duplicated in larger sizes outside of the critical path for extracting distant ILP. On a follow-up paper [13], Kim and Smith discuss the microarchitecture and ISA that implements this distributed processing paradigm, which utilizes hierarchical register files and with a global register file to hold global state.

Another type of factoring is multiclustering [17]. In this approach, the structures of the execution core are split into different execution engines - much like the distributed functional units of [5]. These execution clusters help to scale the instruction scheduling window (including the overall issue bandwidth) and the register file. However, performance can be impacted by cross cluster communication of register values and poor cluster utilization.

Both of these approaches ([13][17]) could be used to further factor the structures of our μ-core, and we will consider this enhancement for future work. However, we consider factoring the structures of the entire processor core, including some popular performance accelerating helper engines, and explore the impact that this factoring has on both performance and power for an aggressive clock.

3. Factored Architectures

The main idea behind factored architectures is to move a set of larger structures out of the regular processor core, resulting in a tiny core with only the necessary components included. Figure 3 shows a conventional(monolithic) core that utilizes a variety of architectural features commonly found in modern microprocessors. These features are necessary to extract as much ILP from the application as possible. However, some of these units such as register file, first level caches, and branch predictor, lie on the critical path of the processor and can therefore have a significant impact on cycle time or critical architectural loops. Figure 3 illustrates our factored architecture, with most of the larger blocks moved out of the critical loops. We demonstrate three different types of factored structures:

- Hierarchical extensions: Caches and branch predictor (shown in light gray)
- Complete factorization: Value predictor and data prefetcher (shown in dark gray)
- Hybrid factorization: Register file and ROB (shown with gray stripes)

Hierarchical extensions remove larger structures from the processor core and leave a much smaller version of the original structure in the core. This benefits structures that are tightly integrated with the processor core for high performance (such as the data cache), particularly those with

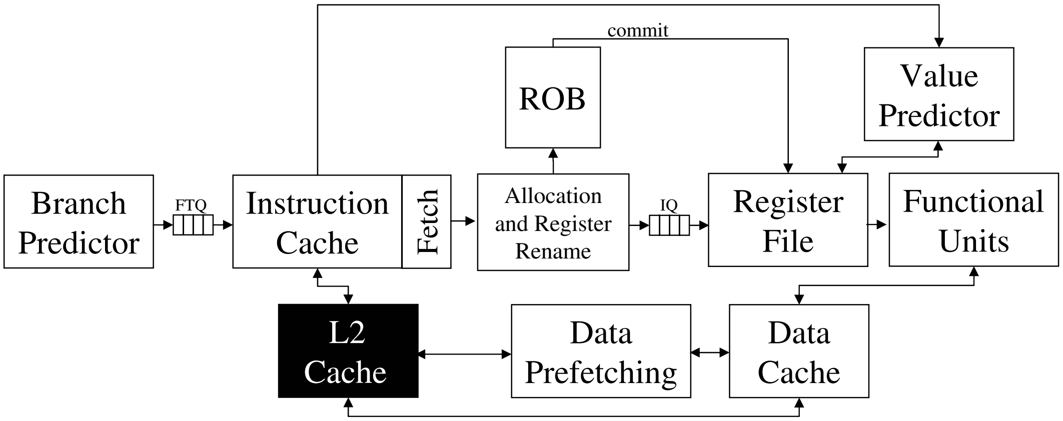

Figure 1: The monolithic core (with all the helper engines are part of the main processor, some of them on the critical path)

high port counts (such as the register file). Complete factorization is suitable for structures with flexible functionality and high latency tolerance that can be configured to require little or no interaction with the core. Some architectural features can use a hybrid of these two approaches (hybrid factorization) - completely factoring some components (e.g. structures of the commit stage) and hierarchically extending others (e.g. the register file).

There have been a number of previous studies on independently factoring individual parts of a processor, as will be introduced in this Section. Yet these studies have not explored what happens when this principle of factoring is applied to more than one structure of the core. We believe our study is unique in synthesizing all of these different ideas together and studying their impact on overall system power.

While structures such as caches are fairly easy to factor, other structures require more extensive analysis and work. In this section we describe how each of the major structures may be factored, and how we model that in our processor simulator.

3.1 Data Cache

We move the L1 data cache out of the core processor pipeline and replace it with a smaller L0 cache, as in [14]. The L0 extends the cache hierarchy, therefore we access the L1 data cache on an L0 miss.

3.2 Instruction Fetch

Similar to the data cache, we insert a smaller L0 instruction cache and move the L1 instruction cache out of the μ-core. To compensate for the smaller cache size, we use out-of-order instruction fetch as described in [18]. In this scheme, a placeholder is used in the instruction fetch queue (IFQ) to maintain program order - and the execution core stalls if the next entry to be consumed from the IFQ is still in flight. We model the complexity this brings to the IFQ by implementing the equivalent of an MSHR [19] for the instruction cache. In the IFQ, we keep the index of the

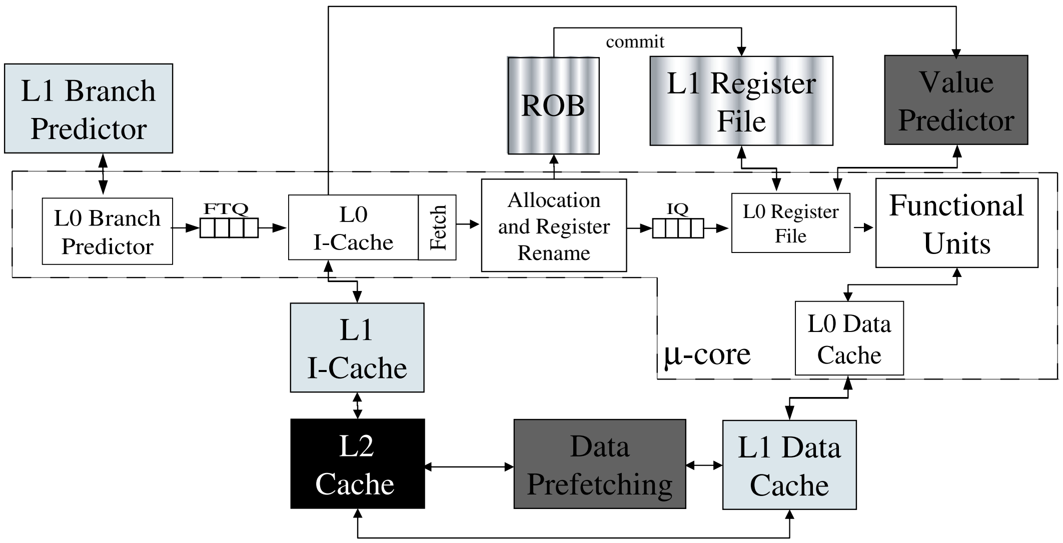

Figure 2: The factored μ-core architecture.

MSHR that will hold the missing instruction cache block along with the instruction's offset inside the cache block. Subsequently, when we see an index to the MSHR in the next IFQ entry to be consumed, we access the MSHR directly to get the required instruction word. The MSHR has a valid bit indicating when the decoding of the instruction can commence. For the case that MSHR runs out of indices or IFQ entries, we stall instruction fetch. However, we found this to be extremely rare without an observable impact on execution performance. Out-of-order fetch is also used in our baseline monolithic case, as it has the additional benefit of reducing instruction cache miss stalls.

3.3 Data Prefetch

We model a stream buffer architecture [20] guided by a stride-filtered markov predictor as proposed in [21]. In the monolithic microprocessor, the address predictors guide prefetches to the stream buffer, which is accessed in parallel with the data cache. When factored, the address predictors and stream buffers are moved further away from the core pipeline. The stream buffers of the factored prefetch engine are *only* accessed on L0 misses. We allow a single prediction and a single prefetch per cycle, guided by the address predictor trained on the L0 miss stream.

3.4 Value Prediction

Value prediction [22] is one approach to break true data dependencies and create more instruction level parallelism in an application. We use a hybrid value predictor [23], yet we only predict load instructions. This structure can be accessed early in the pipeline as we only need the PC of the instruction to make the prediction. Our predictor makes a single prediction per cycle. We make use of an extra bit associated with each instruction in the instruction cache to dynamically mark instructions for value prediction. In this study, we only mark instructions that are loads and that can be confidently predicted by our value predictor.

One way of factoring the value predictor is to store the predicted load value in the register allocated to the load instruction we are predicting. When the data access completes, it overwrites the predicted value in the destination register with the actual load value. If the predicted value and the actual value do not match, a checker engine similar to the ARB [24] detects the misprediction and squashes the mispredicted result and its dependents with the same hardware that is used in a branch misprediction.

While value prediction has not reached popularity in current microprocessor designs, our experimental results indicate that it provides a significant performance improvement for some benchmarks. Our investigation of tuning helper engines for individual applications (Section 5.3) illustrates that for some benchmarks, the value predictor is essential for high performance. For instance, *mcf*, a memory bound application, sees a notable performance improvement with value prediction.

3.5 Branch Address Prediction

Our architecture makes use of a basic block target buffer (BBTB) [25], a branch address predictor that predicts an entire basic block each cycle. The PC at the head of the basic block serves as an index to the predictor, which returns a target address, a fallthrough address, a branch type, and two per-branch prediction counters: one to make per-branch direction predictions, and one to arbitrate between the per-branch prediction and the global branch direction prediction. In the μ-core design, we reduce the size of the BBTB in the core pipeline and add a second level BBTB as done in [15]. Similarly we decouple branch prediction from the instruction cache using fetch target queue (FTQ). On a first level BBTB miss, the second level BBTB is probed and fetch stalls until a response is received from the second level. If the second level misses, we guess a fixed fetch block size and continue fetching until a misprediction is detected.

3.6 Register File and Commit

In the factored architecture we use a multi-level register file similar to the one proposed in [16]. The basic differences are that we model an inclusive register file hierarchy where the second level register file (RF1) includes all the state contained in the first level register file (RF0). Since commit is a vital part of the processor pipeline, this helper engine is never disabled, but the second level register file can be dynamically configured with either 128 or 512 entries at runtime depending on the needs of the application. On a branch misprediction, the second level register file recovers the state of the first level register file. The first level register file releases physical registers when they are no longer the most recent version of a given logical register and when there are no more in-flight instructions that are waiting to consume their value. The second level register file releases a given physical register *pr* that is mapped to a logical register *lr* only after the instruction defining *pr* has committed and a subsequent instruction remapping *lr* has committed. This ensures that the value will not be required for a branch misprediction or to provide precise exceptions. We leverage the fact that the second level register file is only used for commit and branch recovery by factoring the second level register file, reorder buffer (ROB), and commit hardware into a helper engine, similar to [26]. However, our ROB only holds status information for instructions in the pipeline – all result values are stored in the register file, as in the Pentium 4 [27]. Moreover, unlike the future file, our first level register file holds a subset of the total physical registers rather than only the most recent versions of logical registers (effectively holding more state than the future file). In the core pipeline, a simple tag allocator handles ROB allocation, with a tag implicitly mapping to an ROB entry and providing a means to hook up dependent instructions. The register map remains in the core pipeline.

This is a hybrid of complete factorization and hierarchical extension, as the register file is extended with a second level structure, but the commit hardware and ROB are completely factored, with only tag allocation in the ROB impacting the core timing.

4. Methodology

The simulator used in this study was derived from the SimpleScalar/Alpha 3.0 tool set [28], a suite of functional and timing simulation tools for the Alpha AXP ISA. The timing simulator executes only user-level instructions. Simulation is execution-driven, including execution down any speculative path until the detection of a fault, TLB miss, or branch misprediction. Our processor operates at a 5.6GHz clock rate.

We used the SPEC2000 benchmark set for our experiments. Phansalkar et. al. [29], developed a technique that measures the inherent similarity of programs based on their microarchitecture-independent characteristics. Their results show that the SPEC2000 benchmark set can be divided into 8 clusters. Following their approach we show results for a representative subset of the SPEC suite.

The programs were compiled on a DEC Alpha AXP-21164 processor using the DEC C and C++ compilers under OSF/1 V4.0 operating system using full compiler optimization (`-O4 -ifo`). We picked the 4 most dominant phases as determined by the hardware phase detection technique described in [30] and simulated these phases as representative samples of the program. On average, they accounted for approximately 70% of the execution time of each benchmark. All benchmarks were simulated using the *ref* inputs. For each phase, we simulated 30 Million instructions after warming up our architectural structures for 10 Million instructions. The results we report do not include the warmup time for the 10 Million instructions.

4.1 Architectural Model

We have made significant modifications to SimpleScalar to model the various speculative techniques and different configurations in this study. We modeled all of the structures and latencies in the microcore and monolithic architectures.

Table 1 presents the simulation parameters for the monolithic and microcore architectures we explore in this paper. Cache and register file access latencies are extracted from Cacti [31] for a *70nm* Technology at 5.6 GHz frequency.

Note that the difference in branch misprediction penalty is the extra latency attributed to the larger branch predictor, register file and instruction cache in the monolithic core.

4.2 Power and Thermal Simulator

A complete analysis of the static and dynamic power consumption and resulting temperature characteristics of different architectures is crucial to our study. Our power/thermal simulator performs cycle-accurate analysis of investigated architectures based on the following recently developed power and thermal models. We used process parameters for a *70nm* process at 5.6GHz with 1V supply voltage, in order to have a better understanding of next generation submicron, low supply voltage, aggressively clocked microprocessors.

	Monolithic	Microcore	
	Core	L0	Helper Engines
Instruction Window and Physical RF	256 entry ROB 256 entry RF1	128 entry RF0	256 entry ROB 256 entry RF1
BBTB	2048-entry 4-way set associative	256-entry 4-way set associative	2048-entry 4-way set associative
L1 Data Cache	64KB 4-way set associative, dual port with a 32 byte block size, 4 cycle latency	8KB 4-way set associative, dual port with a 32 byte block size, 3 cycle latency	16KB 64-way set associative, single port with a 32 byte block size, 6 cycle latency
L1 Instruction Cache	64KB 2-way set associative, single port with a 32 byte block size, 4 cycle latency	8KB 2-way set associative, single port with a 32 byte block size, 2 cycle latency	64KB 2-way set associative, single port with a 32 byte block size, 5 cycle latency
Value Predictor (1 prediction per cycle)	2K-entry stride 8K-entry markov	none	2K-entry stride 8K-entry L2 markov
Address Predictor (1 prediction per cycle)	2K-entry stride 4K-entry markov	none	2K-entry stride 4K-entry markov
Stream Buffer	32-entry FA buffer	none	32-entry FA buffer
Branch Misprediction	26 cycles	20 cycles	
Core Width	8-way issue, 4-way decode, 4-way commit		
Memory and L2 Cache	150 cycle memory latency, 512KB 4-way set associative unified (instruction and data) cache with a 64 byte block size and 12 cycle latency		
Functional Units	8 integer ALUs, 2 integer MULT/DIV, 2 FP ALU, 2 FP MULT/DIV, 2 load/store		

Table 1: Simulation parameters for the monolithic and microcore architectures.

We have incorporated Wattch [32] models for dynamic power analysis of the microprocessor blocks. The experimental results we present are extracted with the most aggressive conditional clocking strategy, where the dynamic power scales linearly with access to the ports.

For submicron technologies, such as 70nm, leakage power constitutes a significant portion of the overall power. ITRS [33] predicts that leakage power is likely to increase exponentially and make up 50% of the total power dissipation for the next deep submicron processes. Hence, an accurate and reliable leakage power analysis is a necessity. We adapted leakage models from Hotleakage [34] in our power/thermal simulator. Hotleakage models are extended and improved versions of the well-known Butts and Sohi leakage equations [35]. We also used leakage parameters from Hotleakages' pre-determined values specific to the 70nm process technology.

A detailed and accurate thermal analysis of the different architectures we explore in this study is crucial. It has been shown that thermal metrics based on power consumption or power density of individual blocks do not provide accurate thermal estimation [10]. HotSpot [10] provides accurate thermal model that enables more detailed and localized thermal analysis of the microprocessor. It is based on the equivalent circuit of thermal resistance and capacitances, that model the microarchitectural blocks and other aspects of the chip thermal package. We used Hotspot's thermal resistance/capacitance models and RC solvers for our analysis.

Dynamic and leakage power consumption for each microprocessor unit are collected over a predetermined thermal sampling interval, as the temperatures change over periods greater than every cycle. We experimented with various sampling interval lengths in order to explore the trade off between error rate and computational overhead. Hotspot proposes a 10K instruction sampling interval for 180nm and 3.3GHz, our results showed similar error rates for 10K sampling interval for 70nm and 5.6 GHz as well. For our temperature results, we simulate 300 Million instructions after fast-forwarding application-specific number of instructions as proposed in [36].

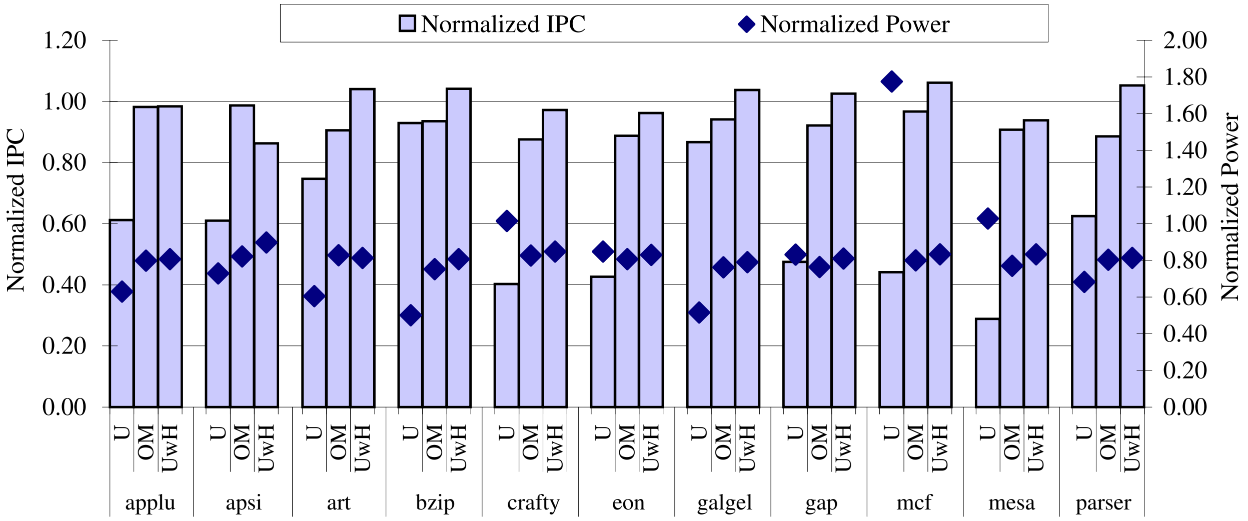

Figure 3: Performance and power comparison of the monolithic core and our μ-core architecture.

Our power/thermal simulator also incorporates the thermal runaway phenomena exposed by the Hotleakage and Hotspot models. Thermal runaway is caused by the exponential dependency of leakage power on temperature: increased temperature increases leakage power, increased leakage power causes an even further increase in temperature. This positive feedback loop between leakage power and temperature is quite significant and can lead to device failure.

5. Results

In this section we evaluate the impact of factored architectures based on the performance and power dissipation and temperature results from our experimental analysis.

5.1 Impact of Factoring

Figure 5.1 compares the performance and power dissipated by three architectures relative to the monolithic core.

- μ-core Architecture (*U*): excludes the first level (L1) data cache, L1 instruction cache, L1 BBTB, value predictor, or data prefetcher. The L2 register file is tuned to hold at most 128 registers. This demonstrates how well the architecture would perform if all structures were simply scaled down without factoring.

- Optimized Monolithic(*OM*): a conventional core with serial access caches, BBTB, and register file. These structures are much more energy efficient, as they pinpoint the specific data array that is required for each access to the structure, consuming energy for that array alone. However, energy savings come at a cost of latency in accessing the structures. We optimistically assume that all structures can be pipelined to accomodate this increased latency without

	L1 DCache	L1 ICache	L2 Cache	Register File	BBTB	Branch Misprediction
Baseline	4 cycle	4 cycle	20 cycle	4 cycle	3 cycle	31 cycle
Optimized	6 cycle	5 cycle	25 cycle	7 cycle	5 cycle	37 cycle

Table 2: Latency parameters for the monolithic and μ-core architectures.

impacting the cycle time of the processor, and further assume that the BBTB can be pipelined using a technique like multiple block ahead prediction [37] without any loss of accuracy.

- μ-core Architecture with Helper Engines (*UwH*): all of the helper engines are activated and the L2 register file is tuned to 512 registers. These configurations are summarized in Table 2.

In Figure 5.1 we present the average of individual phase results, weighted by the amount of execution time contributed by that phase. All results are normalized to the performance of the monolithic core. All architectures make use of simple serial access structures for address and value prediction to save energy.

The optimized monolithic (OM) architecture provides a dramatic power reduction over the baseline monolithic architecture (21% reduction on average). Yet, it can also impact performance due to the added latency of the core structures, dropping IPC by 10% on average.

The main performance benefit of the μ-core (U) over the monolithic architecture comes from the fact that the smaller structures in the μ-core have shorter access latencies, as in the case of the L0 caches. However, for the μ-core architecture, performance is severely impacted by insufficient capacity in the caches and BBTB, insufficient instruction window size, and the lack of value prediction and data prefetching. This drop in performance can actually increase the total power consumed by the μ-core architecture over the baseline monolithic architecture, as seen in `crafty`, `mesa`, and especially `mcf`. The larger power dissipation is due to various reasons, including: an increase in L2 activity due to L0 misses, wasted power on wrong path execution, and simply the overall increase in run time (i.e. a larger amount of leakage power).

However, when all helper engines are enabled, the μ-core architecture is able to achieve higher performance. Even though the structures on the μ-core are much smaller than their monolithic counterparts, they are able to capture enough ILP such that the pipeline remains full while accesses to the larger helper engines are pending. In fact benchmarks such as `art`, `bzip`, `galgel`, `gap`, `mcf`, and `parser` are able to outperform the monolithic case, despite the latency of their helper engines. The benchmark `apsi` experiences a 14% drop in performance due to the reduction in state tracked by the L0 data cache. As will be seen, the miss rate for the first level data cache in `apsi` increases in the μ-core.

Note that the primary impact of the μ-core architecture is on the power consumption of the factored hardware. The fact that it minimizes performance impact while improving power consumption is in stark contrast to most other low-power techniques that try to take advantage of the slack in programs by slowing down the processor. In those schemes, the goal is to maximize power savings with the least amount of slowdown possible. Our results demonstrate that we are able to save as much power as the optimized monolithic (OM) architecture (within 3% on average) yet with performance comparable or better than the baseline monolithic architecture (1% improvement on average).

	IPC	Dcache Miss Rate	Icache Miss Rate	BBTB Miss Rate	BBTB Mispredict Rate
applu	2.00	0.15	0.01	0.00	0.02
apsi	0.96	0.17	0.02	0.00	0.07
art	0.91	0.39	0.00	0.00	0.06
bzip	1.43	0.03	0.00	0.00	0.05
crafty	0.72	0.02	0.06	0.08	0.07
eon	1.03	0.01	0.04	0.04	0.04
galgel	1.83	0.16	0.00	0.00	0.05
gap	1.35	0.03	0.01	0.01	0.06
mcf	0.62	0.42	0.00	0.00	0.05
mesa	1.58	0.01	0.03	0.01	0.04
parser	0.83	0.04	0.00	0.00	0.06

Table 3: Statistics for the baseline monolithic architecture.

	IPC	DL0 Miss Rate	DL1 Miss Rate	IL0 Miss Rate	IL1 Miss Rate	BBTB0 Miss Rate	BBTB1 Miss Rate	BBTB Mispredict Rate
applu	1.95	0.20	0.62	0.04	0.12	0.00	0.15	0.02
apsi	0.83	0.32	0.39	0.04	0.44	0.09	0.58	0.11
art	0.94	0.47	0.69	0.00	0.46	0.00	0.00	0.06
bzip	1.49	0.05	0.43	0.00	0.63	0.00	0.02	0.06
crafty	0.69	0.18	0.11	0.08	0.72	0.28	0.29	0.10
eon	0.97	0.07	0.06	0.07	0.55	0.22	0.20	0.07
galgel	1.90	0.29	0.48	0.00	0.92	0.00	0.00	0.05
gap	1.34	0.04	0.25	0.04	0.46	0.19	0.05	0.07
mcf	0.67	0.44	0.85	0.00	0.59	0.00	0.00	0.05
mesa	1.46	0.11	0.03	0.09	0.28	0.23	0.05	0.07
parser	0.88	0.09	0.40	0.02	0.04	0.02	0.05	0.07

Table 4: Statistics for the μ-core architecture with all helper engines enabled.

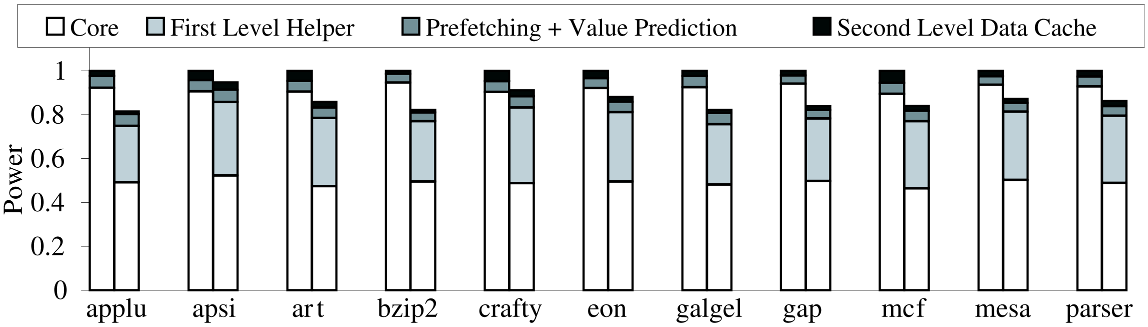

Figure 4: Power breakdown for Monolithic and μ-core architectures. (Normalized by monolithic architecture power for each benchmark)

Figure 5.1 shows how different helper engines contribute to power consumption in our μ-core architecture. Power values are normalized to the baseline monolithic architecture (shown in the first bar). The second bar represents the power of the μ-core with all helper engines enabled. We separated value prediction and prefetching from other helpers, because they are completely factored. *First level helper* accounts for power consumed in all the other helper engines including the DL1, IL1, BBTB1, and the hierarchically extended register file.

Figure 5.1 also illustrates the reduction in core power in the μ-core compared to the baseline monolithic architecture. The core dissipates 90-94% of the power in baseline monolithic architecture, compared to 54-60% in the μ-core architecture. With more power dissipated in the latency tolerant helper engines, the μ-core is able to take better advantage of optimizations trading performance for power.

Further data is presented in Tables 3 and 4. Despite an increase in the L0 miss rates for the hierarchically extended structures (note that the register file design does not miss), the μ-core with helper engines (UwH) is still able to reduce the power impact with a slight average IPC improvement.

5.2 How Far Can We Factor?

To better explore how far we can scale the size L0 structures without performance loss, we simulated a fairly large design space of structure sizes for the hierarchically extended helper engines.

Figure 5.2 presents results for different data cache sizes and associativities. We adjusted latencies for each configuration using CACTI [31]. While smaller associativities seem to help reduce power dissipation, associativity is critical for high performance in applications like `apsi` and `applu`. Capacity is also important – applications such as `art` and `galgel` see a degradation in performance when going to smaller sized caches, even with comparable associativities.

Results for different BBTB sizes (in number of entries) and associativities are illustrated in Figure 5.2. Although some benchmarks are unaffected by reduced BBTB state, others such as `crafty`, `eon`, `gap`, `mesa`, and `parser` see dramatic degradation in performance as overall predictor size or associativity are decreased. Despite available backing storage from the helper engines, there is

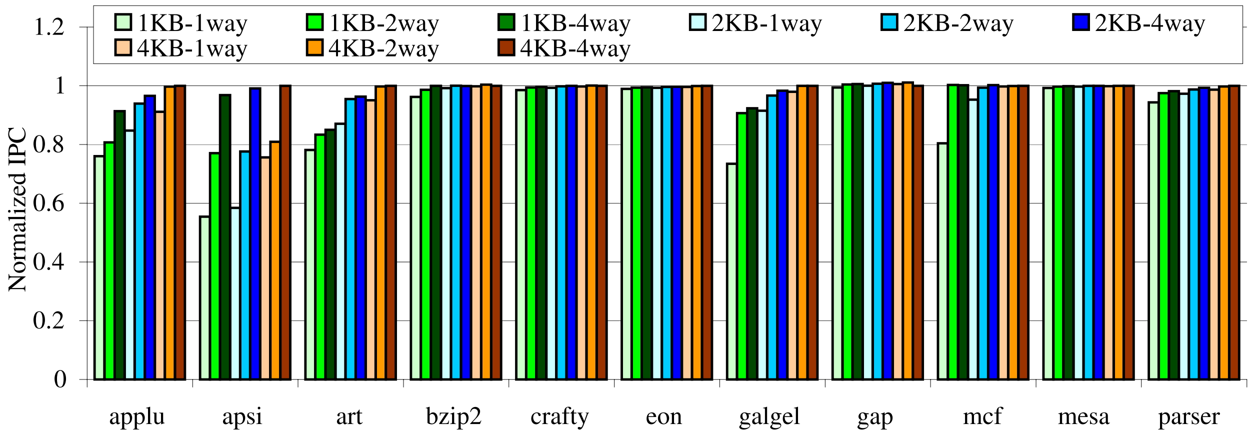

Figure 5: IPC impact of L0 data cache scaling.

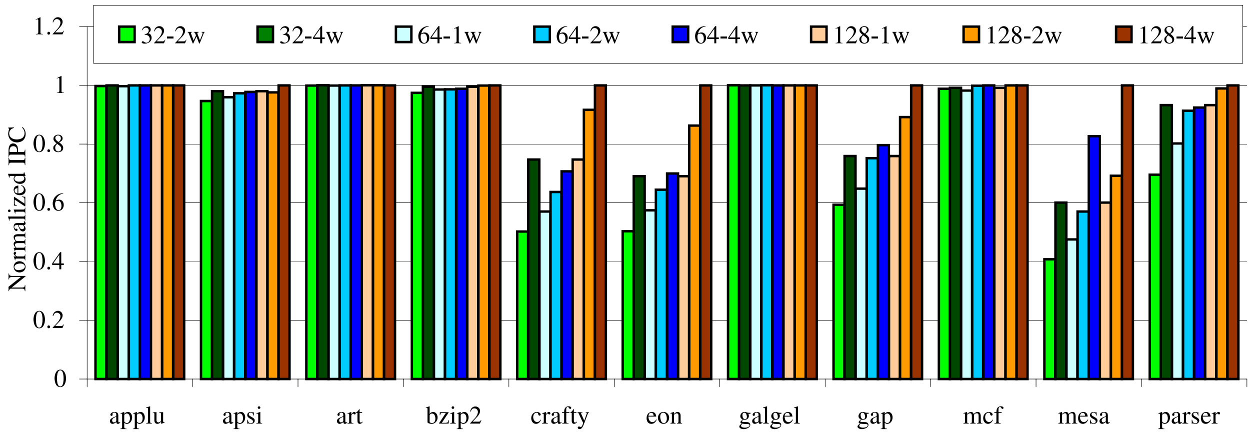

Figure 6: IPC impact of L0 BBTB scaling.

clearly a critical amount of state that must be maintained in the core to achieve high performance. We skip the results for the L0 instruction cache size, as almost of the benchmarks seem to be tolerant of that. Only `mesa` saw any significant IPC degradation, even for 1KB instruction caches.

5.3 Tuning Helper Engines

The results presented thus far, have assumed that all of the helper engines are on at all times. As discussed in Section 2, programs can be divided into blocks of similar execution behavior, called phases. Various techniques have been proposed to extract the phase behavior of a program. Program phase behaviour can provide even more significant energy savings by allowing us to selectively turn on/off a subset of the available helper engines. Note that it would be possible to dynamically tune the monolithic architectural structures to a particular program phase (as in [38]), but this approach limits how far the latency of these structures can be reduced, as the structures still consume the same chip area whether they are on or off, and can still impact wire routing and wire latency. Tuning the helper engines is one alternative to this, and will migrate the control logic needed to perform the reconfiguration and the actual larger structures themselves away from the core. The μ-core structures could still be dynamically tuned as in prior work to provide even more power savings. The flexibility of helper engine design allows us to independently control individual units so that only the critical helpers for the execution of a given application phase are actually on, thus reducing overall power dissipation. In the μ-core design, there are number of helper engines (our initial implementation has six) all working together to speed up the execution of a program. The key observation here is that for any given program phase, some helpers are more helpful than others, yet in the simplest design they are all being used and all consuming power. In many cases it is possible to turn off the non-critical helper engines to reduce the power consumption. To achieve this, we need a policy for turning helper engines on/off or tuning them (e.g. in the case of the L2 register file).

However it is not very straight-forward to determine which helper engines should be turned on/off to minimize the power for a given performance target. The performance improvement due to each helper engine is *not* independent from the status of the other helper engines. For example, using only the large branch predictor or the large instruction cache provides no additional performance, yet by using both there might be significant improvement. This is a fairly simple example, the real interactions between helpers like value prediction and prefetching are far more complex. This part of the experiments is targeted to investigate the above effect.

Figure 5.3 illustrates the optimal configuration (in $\frac{Energy}{IPC^2}$) for the four most dominant phases of each benchmark. The columns of the table represent the helper engines (r=ROB, v=value prediction, b=branch prediction, i=instruction cache, p=prefetching, c=data cache). A marked square indicates that for the given phase the corresponding helper engine should be“on” for best performance.

Helper engine configurations usually refer to “on” or “off”. However, the L2 register file is never turned off, but is tuned so that either 128 or 512 entries in the file are active and available for renaming. The final two columns show the normalized IPC and total power dissipation for each configuration relative to keeping all helper engines on. The configurations in Figure 5.3 all achieve performance within 5% of the highest performance configuration. They were found by a brute force search of the design space, simulating every possible case and taking the configuration with the lowest $\frac{Energy}{IPC^2}$ measurement. It is important to observe in Figure 5.3 that there is no *one* good

		r	v	b	i	p	c	IPC	Power
applu	1	X			X	X	X	0.97	0.92
	2		X		X	X	X	0.97	0.76
	3	X			X	X	X	0.95	0.92
	4				X	X	X	0.95	0.71
art	1					X	X	0.98	0.64
	2					X	X	0.97	0.65
	3					X	X	0.96	0.64
	4					X	X	0.97	0.65
crafty	1			X	X		X	0.98	0.63
	2			X	X		X	0.99	0.63
	3			X	X		X	0.99	0.63
	4			X	X		X	0.98	0.63
galgel	1	X						0.97	0.86
	2					X	X	0.97	0.63
	3	X					X	0.96	0.87
	4	X					X	0.96	0.87
mcf	1		X			X	X	0.98	0.71
	2		X				X	0.98	0.67
	3		X			X	X	0.97	0.71
	4		X				X	0.98	0.67
parser	1		X	X	X		X	0.98	0.68
	2		X	X	X	X	X	0.96	0.74
	3			X	X		X	0.96	0.64
	4		X	X	X	X	X	0.96	0.73

		r	v	b	i	p	c	IPC	Power
apsi	1				X	X	X	0.96	0.70
	2			X	X	X	X	0.98	0.71
	3				X	X	X	0.96	0.70
	4			X	X		X	0.98	0.65
bzip2	1					X		0.95	0.61
	2		X	X	X	X	X	0.96	0.72
	3					X		0.96	0.60
	4						X	1.00	0.56
eon	1			X	X		X	0.99	0.64
	2			X	X		X	0.99	0.63
	3			X	X		X	0.98	0.63
	4			X	X		X	0.97	0.64
gap	1			X	X		X	0.96	0.63
	2		X	X	X		X	0.97	0.71
	3			X	X		X	0.97	0.63
	4		X			X		1.00	0.70
mesa	1			X	X		X	0.97	0.63
	2			X	X		X	0.97	0.64
	3			X	X		X	0.97	0.64
	4			X	X		X	0.97	0.64

Figure 7: Optimal configuration.

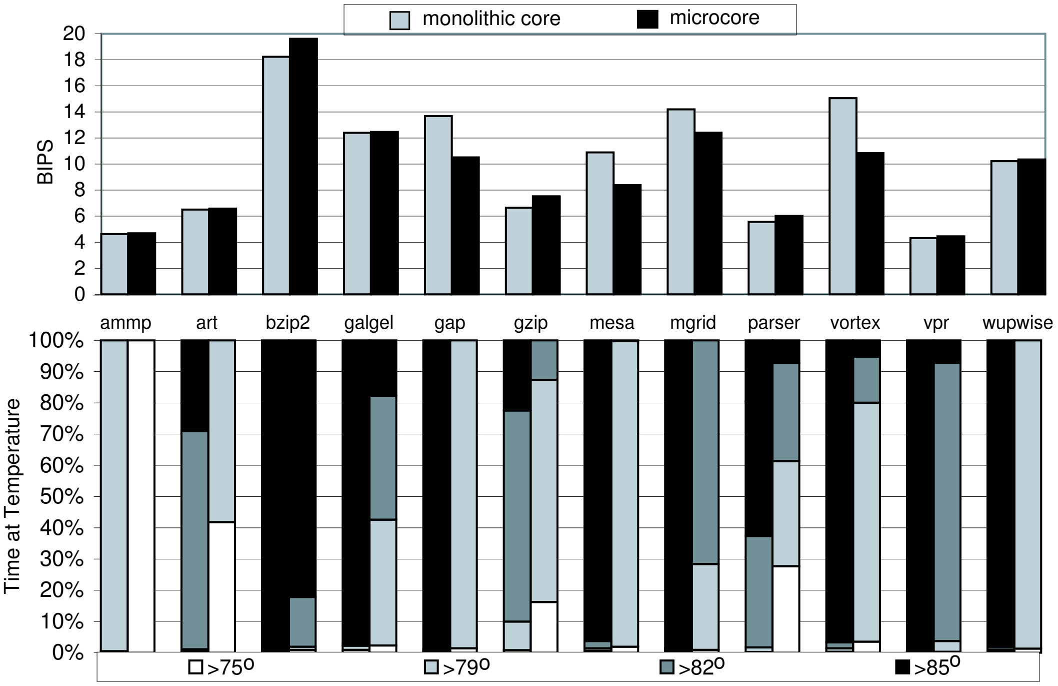

Figure 8: Performance and thermal behavior of a microcore(μ-core) vs a monolithic core

configuration, but rather there are multiple good configurations for each benchmark. Furthermore for some benchmarks very different configurations are best suited for different phases.

Obviously, a brute force search among exponential number of possible cases is not a good design choice for a runtime system. Therefore we include hardware use counters along with each helper engine to determine the utility of each structure. For each program phase (dynamically identified as in [30]), the utility of each structure can be determined after the initial execution of that phase, where the usage counts of each helper engine determines whether or not that engine should be activated in that phase. Usage counters will only be incremented when a helper engine is actually successfully used, not just probed (i.e. the number of L1 data cache hits would be counted, not the number of accesses). Each helper engine would have an activation threshold that the usage counter would need to exceed before it is considered necessary for that phase. This dynamically formed phase configuration can be stored in a predictor and used guide future configuration of the helper engines. However, such an approach is not guaranteed to find an optimal configuration. Future work will explore such approaches and more intelligent schemes to selectively enabling helper engines.

5.4 Thermal Characteristics of a Microcore vs a Monolithic Core

Figure 5.4 compares the performance and thermal behavior of a conventional monolithic core and the μ-core architecture on some of the SPEC 2000 benchmarks. The upper half of the figure shows performance in BIPS for different benchmarks, and the lower half illustrates the heating behavior of the investigated architectures. This latter component shows the percentage of cycles for which at least one block exceeds the indicated temperatures: 75°C, 79°C, 82°C and 85°C. Darker colors in the lower graphs indicate higher temperatures. The rest of the figures in this section are similarly constructed. For example, galgel sees comparable performance with either the μ-core or monolithic architecture, but the monolithic core sees a temperature greater than 85°C almost 97% of the time. The μ-core only exceeds 85°C around 18% of the time, and stays below 82°C around 42% of the time. Note that for many benchmarks, and particularly in monolitic architectures, temperature frequently exceeds the thermal threshold, 82°C. These results should be considered as an upper bound for performance that are not be achievable without some form of thermal management. On-chip temperatures for the μ-core architecture are significantly lower than the monolithic core, but it still retains good performance comparable to that of the monolithic core. This can be attributed to the significantly smaller structures in the μ-core that are much more power efficient. It is important to note that the ITRS projects a reduction in maximum permited junction temperatures for the future generations of process technologies. The maximum tolerated junction temperatures are around 85°C for 130nm and even lower for smaller process technologies.

6. Conclusion

In this paper, we have proposed a factored architecture that is enhanced by a number of different helper engines. The critical pipeline structures of instruction fetch, branch prediction, data memory access, and the register file have all been hierarchically extended, and the value predictor, data prefetch engine, and commit hardware have been completely decoupled from the factored core. This new microcore architecture is able to reduce total processor power dissipation by 20% on average, while it attains comparable or better performance than a deeply pipelined monolithic design at the same clock frequency. This power gain does not take into account further possible reduction in routing complexity and wire delay that can be achieved by reducing the size of structures in the critical pipeline core. The flexibility of the microcore architecture opens the door to further energy-saving techniques that can be applied to the more latency tolerant helper engine structures. Furthermore, through dynamic configuration of the helper engines to different application phases, an additional 13% power savings can be attained with only an average 3% degradation in performance.

We demostrated that the microcore architecture provides lower on-chip temperatures compared to a conventional monolithic architecture. Factoring larger and power-hungry structures out of the core limits the number of accesses to such blocks and reduces the amount of heat generated by them. Our experimental results show that the microcore reduces the number of cycles over the critical thermal threshold by 86% on average, even without any dynamic thermal management technique.

References

[1] V. Agarwal, M. Hrishikesh, S. Keckler, and D. Burger, "Clock rate versus ipc: The end of the road for conventional microarchitectures," in *27th Annual International Symposium on Computer Architecture*, 2000.

[2] E. Borch, E. Tune, S. Manne, and J. Emer, "Loose loops sink chips," in *Proceedings of the Eighth International Symposium on High-Performance Computer Architecture*, 2002.

[3] E. Sprangle and D. Carmean, "Increasing processor performance by implementing deeper pipelines," in *29th Annual International Symposium on Computer Architecture*, 2002.

[4] D. Brooks, P. Cook, P. Bose, S. Schuster, H. Jacobson, P. Kudva, A. Buyuktosunoglu, J. Wellman, V. Zyuban, and M. Gupta, "Power-aware microarchitecture: Design and modeling challenges for next-generation microprocessors," in *IEEE Micro*, Nov. 2000.

[5] J. E. Smith, "Instruction-level distributed processing," *IEEE Computer*, vol. 34, pp. 59–65, Apr. 2001.

[6] T. Mudge, "Power: A first class design constraint for future architecture and automation," in *Proceedings of the 7th International Conference on High Performance Computing*, 2000.

[7] S. Gunther, F. Binns, D. Carmean, and J. Hall, "Managing the impact of increasing microprocessor power consumption," in *Intel Technology Journal Q1*, 2001.

[8] D. Brooks and M. Martonosi, "Adaptive thermal management for high-performance microprocessors," in *Workshop on Complexity Effective Design*, June 2000.

[9] D.Brooks and M.Martonosi, "Dynamic thermal management for high-performance microprocessors," in *International Symposium on High-Performance Computer Architecture (HPCA-7)*, pp. 171–182, Jan. 2001.

[10] K.Skadron, M.Stan, W. Huang, S.Velusamy, K. Sankaranarayanan, and D. Tarjan, "Temperature-aware microarchitecture," in *30th Annual International Symposium on Computer Architecture*, pp. 2–13, June 2003.

[11] W.Huang, J.Renau, S-M.Yoo, and J. Torrellas, "A framework for dynamic energy effiency and temperature management," in *33rd International Symposium on Microarchitecture*, pp. 202–213, Dec. 2000.

[12] C.-H. Lim, W. Daasch, and G.Cai, "A thermal-aware superscalar microprocessor," in *International Symposium on Quality Electronic Design*, pp. 517–522, Mar. 2002.

[13] H.-S. Kim and J. E. Smith, "An instruction set and microarchitecture for instruction level distributed processing," in *Proceedings of the 29th annual international symposium on Computer architecture*, pp. 71–81, June 2002.

[14] J. Kin, M. Gupta, and W. Mangione-Smith, "The filter cache:an energy efficient memory structure," in *IEEE International Symposium on Microarchitecture*, Dec. 1997.

[15] G. Reinman, T. Austin, and B. Calder, "A scalable front-end architecture for fast instruction delivery," in *26th Annual International Symposium on Computer Architecture*, May 1999.

[16] R. Balasubramonian, S. Dwarkadas, and D. Albonesi, "Reducing the complexity of the register file in dynamic superscalar processors," in *Proceedings of the 34th Annual International Symposium on Microarchitecture*, Dec. 2001.

[17] K. Farkas, P. Chow, N. Jouppi, and Z. Vranesic, "The multicluster architecture: Reducing cycle time through partitioning," in *International Symposium on Microarchitecture*, 1997.

[18] J. Stark, P. Racunas, and Y. N. Patt, "Reducing the performance impact of instruction cache misses by writing instructions into the reservation stations out-of-order," in *30th International Symposium on Microarchitecture*, pp. 34–43, Dec. 1997.

[19] D. Kroft, "Lockup-free instruction fetch/prefetch cache organization," in *8th Annual International Symposium of Computer Architecture*, pp. 81–87, May 1981.

[20] N. Jouppi, "Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers," in *Proceedings of the 17th Annual International Symposium on Computer Architecture*, May 1990.

[21] T. Sherwood, S. Sair, and B. Calder, "Predictor-directed stream buffers," in *33rd International Symposium on Microarchitecture*, Dec. 2000.

[22] M. Lipasti and J. Shen, "Exceeding the dataflow limit via value prediction," in *Proceedings of the 29th Annual ACM/IEEE International Symposium on Microarchitecture*, December 1996.

[23] K. Wang and M. Franklin, "Highly accurate data value prediction using hybrid predictors," in *30th Annual International Symposium on Microarchitecture*, pp. 281–290, Dec. 1997.

[24] M. Franklin and G. S. Sohi, "Arb: A hardware mechanism for dynamic reordering of memory references," *IEEE Transactions on Computers*, vol. 46, May 1996.

[25] T. Yeh and Y. Patt, "A comprehensive instruction fetch mechanism for a processor supporting speculative execution," in *Proceedings of the 25th Annual International Symposium on Microarchitecture*, pp. 129–139, Dec. 1992.

[26] J. E. Smith and A. R. Pleszkun, "Implementation of precise interrupts in pipelined processors," in *12th Annual International Symposium on Computer Architecture*, 1985.

[27] G. Hinton, D. Sager, M. Upton, D. Boggs, D. Carmean, A. Kyker, and P. Roussel, "The microarchitecture of the pentium 4 processor," *Intel Technology Journal Q1*, 2001.

[28] D. C. Burger and T. M. Austin, "The simplescalar tool set, version 2.0," Technical Report CS-TR-97-1342, U. of Wisconsin, Madison, June 1997.

[29] A.Phansalkar, A.Joshi, L.Eeckhout, and L.Kohn, "Measuring program similarity: Experiments with spec cpu benchmark suites," pp. 2–13, Mar. 2005.

[30] T. Sherwood, S. Sair, and B. Calder, "Phase tracking and prediction," in *30th Annual International Symposium on Computer Architecture*, June 2003.

[31] P. Shivakumar and N. P. Jouppi, "Cacti 3.0: An integrated cache timing, power, and area model," in *Technical Report*, 2001.

[32] D. Brooks, V. Tiwari, and M. Martonosi, "Wattch: A framework for architectural-level power analysis and optimization," in *27th Annual International Symposium on Computer Architecture*, pp. 83–94, June 2000.

[33] in *International Technology Roadmap for Semiconductors*, 2003.

[34] Y. Zhang, D. Parikh, K. Sankaranarayanan, K. Skadron, and M. Stan, "Hotleakage: A temperature-aware model of subthreshold and gate leakage for architects," in *University of Virginia Dept of Computer Science Tech Report CS-2003-05*, Mar. 2003.

[35] J. Butts and G. Sohi, “A static power model for architects,” in *27th Annual International Symposium on Computer Architecture*, pp. 191–201, June 2000.

[36] T. Sherwood, E. Perelman, G. Hamerly, and B. Calder, “Automatically characterizing large scale program behavior,” in *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems*, Oct. 2002.

[37] A. Seznec, S. Jourdan, P. Sainrat, and P. Michaud, “Multiple-block ahead branch predictors,” in *Proceedings of the Seventh International Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 116–127, Oct. 1996.

[38] D. Albonesi, “Selective cache ways: On-demand cache resource allocation,” in *32nd International Symposium on Microarchitecture*, Nov. 1999.