The Design and Implementation of Heterogeneous Multicore Systems for Energy-efficient Speculative Thread Execution

YANGCHUN LUO, Advanced Micro Devices Inc.
WEI-CHUNG HSU, National Chiao Tung University, Taiwan
ANTONIA ZHAI, University of Minnesota, Twin Cities

With the emergence of multicore processors, various aggressive execution models have been proposed to exploit fine-grained thread-level parallelism, taking advantage of the fast on-chip interconnection communication. However, the aggressive nature of these execution models often leads to excessive energy consumption incommensurate to execution time reduction. In the context of Thread-Level Speculation, we demonstrated that on a same-ISA heterogeneous multicore system, by dynamically deciding how on-chip resources are utilized, speculative threads can achieve performance gain in an energy-efficient way.

Through a systematic design space exploration, we built a multicore architecture that integrates heterogeneous components of processing cores and first-level caches. To cope with processor reconfiguration overheads, we introduced runtime mechanisms to mitigate their impacts. To match program execution with the most energy-efficient processor configuration, the system was equipped with a dynamic resource allocation scheme that characterizes program behaviors using novel processor counters.

We evaluated the proposed heterogeneous system with a diverse set of benchmark programs from SPEC CPU2000 and CPU20006 suites. Compared to the most efficient homogeneous TLS implementation, we achieved similar performance but consumed 18% less energy. Compared to the most efficient homogeneous uniprocessor running sequential programs, we improved performance by 29% and reduced energy consumption by 3.6%, which is a 42% improvement in energy-delay-squared product.

Categories and Subject Descriptors: C.1.2 [**Processor Architectures**]: Multiple Data Stream Architectures (Multiprocessors)—*Multiple-instruction-stream, multiple-data-stream processors (MIMD)*

General Terms: Design, Experimentation, Performance

Additional Key Words and Phrases: Thread-Level Speculation, energy efficiency, heterogeneous multicore, dynamic resource allocation

ACM Reference Format:
Luo, Y., Hsu, W.-C., and Zhai, A. 2013. The design and implementation of heterogeneous multicore systems for energy-efficient speculative thread execution. ACM Trans. Architec. Code Optim. 10, 4, Article 26 (December 2013), 29 pages.
DOI: http://dx.doi.org/10.1145/2541228.2541233

1. INTRODUCTION

One primary research objective pertaining to computer systems is to improve performance. In the past few decades, this objective had been largely realized through

This work is supported in part by a grant from the National Science Foundation under CNS-0834599, CCF-0916583, and CPS-0931931; a contract from Semiconductor Research Cooperation under SRC-2008-TJ-1819; and a gift grant from IBM.
Authors' addresses: Y. Luo, Advanced Micro Devices, One AMD Place, Sunnyvale, CA 94088, email: Yangchun.Luo@amd.com; W.-C. Hsu, National Chiao Tung University, Taiwan; email: hsu@cs.nctu.edu.tw; A. Zhai, University of Minnesota - Twin Cities; email: zhai@cs.umn.edu.

DOI: http://dx.doi.org/10.1145/2541228.2541233

the efforts in increasing clock frequency and exploiting instruction-level parallelism (ILP). The pursuit along this path has hit a major stall since the beginning of the last decade. The power density issue has blocked the continuous increase in clock speed. As device size shrinks, cross-chip wire latency has made the highly interconnected designs infeasible. Moreover, it has been observed that further exploiting ILP returns diminishing improvement. As a result, the computer industry has shifted to building multi-threaded processor architectures that exploit thread-level parallelism (TLP).

Automatic TLP extraction by the compiler is desirable but is challenging due to ambiguous memory aliasing, and its effectiveness is quite limited for general-purpose applications. In the presence of complex control flow and ambiguous pointer usage, traditional parallelization schemes must conservatively ensure correctness by synchronizing all potential dependences, which inevitably limits parallel performance. Novel execution models such as Thread-Level Speculation (TLS) have been proposed to aggressively extract thread-level parallelism with minimal manual intervention. In the TLS model, the compiler is allowed to parallelize a sequential program without the burden to first prove independence among the extracted threads. All potential dependences can be speculatively ignored at compile time. During runtime, dependence violation is detected and recovered by built-in hardware support. With speculation, potential TLP that is otherwise difficult to extract could be exploited without any manual effort in modifying the source code.

TLS execution has both advantages and disadvantages. When speculation is successful, it improves performance and reduces the duration in which static power is being consumed. However, when speculation fails, speculative execution consumes more dynamic power by performing activities whose results are eventually thrown away and more leakage power since more on-chip components are kept active. Thus, eliminating ineffective speculation is the first step to improving energy efficiency for TLS. A number of works have investigated both static [Liu et al. 2006; Johnson et al. 2004; Vijaykumar and Sohi 1998; Sohi et al. 1995; Johnson et al. 2007] and dynamic [Luo et al. 2009; Luo and Zhai 2012] optimization techniques to avoid ineffective speculations.

In this article, we propose a same-ISA *heterogeneous* multicore architecture that achieves performance gain in an energy-efficient way. Since energy consumption of speculative threads is highly dependent on their runtime behaviors and the underlying microarchitecture, statically allocating threads to optimize energy efficiency is difficult. Furthermore, we have found that, within the same application, the most energy-efficient processor configuration varies as the program enters different segments of execution. Thus, it is crucial to *dynamically* characterize speculative thread behaviors and configure the processor to match their respective characteristics.

1.1. Different Forms of Heterogeneity

Integration of heterogeneous cores onto a single die has been proposed previously for executing sequential [Kumar et al. 2003] and parallel [Kumar et al. 2004] workloads. These techniques, however, cannot be directly translated to the speculative-thread environment. We intend to consider a set of same-ISA cores with different computing power (i.e., issue width) and with or without hardware support for Simultaneous Multithreading (SMT).

Cache configurations have significant impact on energy efficiency. The cache components can be configured in terms of total size, number of sets, and associativity. Reconfiguring cache for speculative threads exhibits complexities that do not exist in sequential programs. In this article, we only focus on the impact of first-level cache(s) and hold second-level cache size (which is also the last-level cache on-chip) constant. This is because all cores share the same second-level cache in our system; the aggregated memory accesses generated by all speculative threads are similar to that

Table I. Guidelines for Resource Allocation

Guideline I	Prefer smaller superscalar cores or an SMT core when ILP is low.
Guideline II	Prefer SMT when data is shared at a fine granularity.
Guideline III	Speculation failure can be benign if it prefetches data.
Guideline IV	Prefer CMP if processor resources are highly contended.
Guideline V	Carefully manage resources wrt. the reconfiguration overhead.

generated by the corresponding sequential program. Thus, altering the second-level cache size will have similar impact on performance whether or not the program is speculatively parallelized.

TLS can be supported on processors with different forms of multithreading: a single core with SMT support, as well as multiple independent cores (Chip Multiprocessor [CMP]) each executing a single thread. Previous work has shown that SMT and CMP could result in different levels of energy efficiency [Packirisamy et al. 2008]. Thus, we explore the possibility for both types of multithreading support.

1.2. Dynamic Resource Allocation Guidelines

Speculative threads exhibit a number of unique sharing and contention patterns, which pose as both challenges and optimization opportunities for our design. The resource allocation guidelines for each case are summarized in Table I.

1.2.1. Tradeoff between Instruction-Level and Thread-Level Parallelism. When a program executes without being speculatively parallelized, ILP can be extracted from the original sequential execution trace. When parallelized with speculative threads, each thread carries a much shorter instruction trace, which in turns leaves less instruction-level parallelism for the processor to exploit. In this case, smaller superscalar cores or a single SMT core is preferred.

1.2.2. Data Spread across Multiple L1 Caches. Speculative threads extracted from sequential programs often share data at a fine granularity. When parallelized across multiple superscalar cores, each with its own private L1 cache, the average L1 cache miss rate can increase due to the fact that data must be loaded into multiple L1 caches. These additional cache misses do not occur when parallel threads execute on a single core that supports SMT. Therefore, SMT is preferred when data is shared at a fine granularity.

1.2.3. Prefetching Effect between Speculative Threads. Data brought into the shared cache by speculative threads can also be used by other speculative or nonspeculative threads, thus eliminating cache misses that would otherwise occur. One difference between speculative threads and fully parallel threads is that speculative threads have an implied order while parallel threads may be executed in any order. Due to the ordering constraint, speculative threads have been particularly effective in prefetching data into the shared cache for the subsequent threads [Luo et al. 2009]. Even a failed speculative thread may still improve performance without degrading energy efficiency.

1.2.4. Resource Contention among Speculative Threads. Speculative threads share resources when executed in an SMT processor. If all the threads are computationally intensive, the resource contention can stifle performance that can be otherwise achieved in a CMP. This situation can be exacerbated when speculative threads compete for resources but eventually fail and commit no useful work. Thus, we should favor CMP if resources are highly contended among speculative threads.

1.2.5. Frequent Code Segment Interleaving with Varying Performance Characteristics. TLS, taking advantage of fast on-chip communication latency, can potentially improve performance even when parallel threads are very small. This phenomenon leads to the frequent interleaving of parallel and sequential segments, as well as parallel segments with different performance characteristics. Consecutive code segments may require completely different architecture configurations to reach their best energy efficiency; however, the reconfiguration overhead can completely eliminate the benefit. Thus, we must carefully manage thread allocation and thread migration.

1.3. Contributions

This work aims to improve energy efficiency of speculative execution by proposing a heterogeneous multicore system. The following contributions are made:

—We evaluate the energy efficiency of speculative threads on diverse architectural configurations and identify a subset that when integrated as a heterogeneous multicore system can enable energy-efficient execution for a large percentage of the programs.
—We identify and evaluate various overheads associated with speculative thread migration and cache reconfiguration in a heterogeneous multicore system and propose mitigation mechanisms that can drastically reduce their impacts.
—We propose a dynamic resource allocation scheme that analyzes the characteristics of speculative thread execution and identify an energy-efficient configuration on a heterogeneous multicore system for each segment of execution.

The rest of this article is organized as follows: Section 2 describes the experiment and evaluation infrastructures. We discuss how to build the heterogeneous multi-core system in Section 3 and also estimate the potential in energy efficiency improvement. Section 4 deals with two vital issues of system implementation: overhead mitigation and resource allocation. We evaluate the proposed system in Section 5. Related work is addressed in Section 6, and the article is concluded in Section 7.

2. EXPERIMENT INFRASTRUCTURE

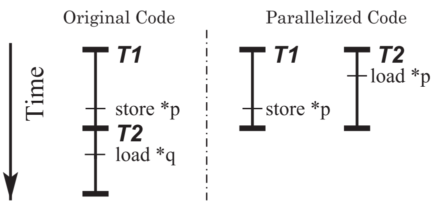
In this section, we will present the details on the speculative thread execution model as well as our evaluation infrastructures.

2.1. Speculative Thread Execution Model

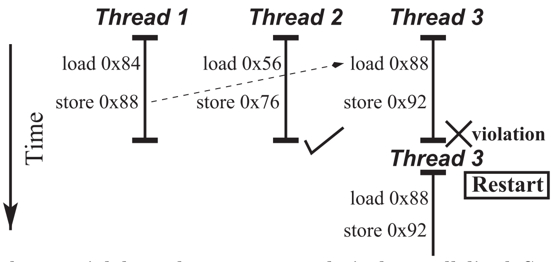
The TLS model allows the compiler to parallelize a sequential program without first proving the independence among the extracted threads. During runtime, the underlying hardware keeps track of each memory access, determines whether any data dependence is violated, and re-executes the violating thread(s) as needed. Figure 1 contrasts the TLS execution model with the traditional parallelization scheme.

In Figure 1(a), the compiler attempts to partition the sequential execution into two parallel threads `T1` and `T2`. As a result, two pointer-based memory accesses, a logically earlier store and a logically later load, are allocated to two different threads, and their relative order is inverted in parallel execution. This partitioning is safe only if these two instructions access different memory locations. However, their memory addresses are unknown at the compilation time. Hence, the compiler is forced to give up parallelizing this code region. Traditional compilers have very limited capability in parallelizing pointer-based codes.

Figure 1(b) illustrates the concept of speculative execution. The threads are numbered according to their original sequential order. If no dependence is violated, the speculative thread commits (`thread 2`); otherwise, it is squashed and re-executed (`thread 3`). TLS empowers compilers to parallelize program regions that were previously nonparallelizable.



(a) The relative order of memory accesses can be inverted after parallelization. Compiler cannot parallelize this segment of code due to possible memory aliasing.



(b) Code regions with potential dependences are speculatively parallelized. Speculation succeeds in thread 2 and fails in thread 3.

Fig. 1. TLS can parallelize sequential code regions that are otherwise difficult to parallelize by traditional methods.

2.2. Architectural Support for Speculation

Speculative threads can be supported on a single core with SMT extension (SMT-based) or on multiple cores each running one thread (CMP-based).

We adopted the STAMPede approach to support TLS on a CMP processor [Steffan et al. 2000, 2005]. All processing cores have private first-level caches that are connected through a bus to the shared second-level cache. The STAMPede approach extends the cache coherence protocol with two new states, speculatively shared (*SpS*) and speculatively exclusive (*SpE*), and transitions to and from these states. All the speculative threads are assigned with a unique ID that determines the logic order in which the threads commit. The thread ID of the sender piggybacks on all invalidation messages. If a cache line is speculatively loaded, it enters the *SpS* or *SpE* state and becomes susceptible to dependence violations. If an invalidation message arrives from a logically earlier thread for that cache line, speculation fails, and the thread is squashed. To recover a speculation failure, all cache lines in the *SpE* state are invalidated and all *SpS* lines transit into the shared state; the thread is then re-executed from the beginning. When a thread is spawned to a busy core that is executing an earlier thread, the new thread is suspended and will be resumed when the earlier thread commits.

The SMT architecture is based on the proposal by Lo et al. [1997], where processor resources are fully shared by all threads. SMT cores are extensions of superscalar cores of the same issue width, except that renaming unit and register files are replicated. Up to two threads are allowed to fetch instructions in the same cycle based on the ICOUNT fetch policy [Tullsen et al. 1996]. Speculative writes are buffered at the first-level cache, which is shared by all the SMT threads. No bus transaction is required for interthread communications. Each first-level cache line is further extended with bits to indicate whether it is speculatively loaded or modified, and if so, by which thread(s). The rest of the hardware support and implementation details can be found in Packirisamy et al. [2006].

Table II. Default Processor Parameters

Pipeline Stages Fetch/Issue/Commit Width	Fetch/Issue/Ex/WB/Commit 6/4/4
ROB/LSQ Size	128/64 entries
Integer Units	6 units/1 cycle
Floating Point Units	4 units/12 cycles
Private Level 1 Data Cache	64 KB, 4-way, 32 B
Private Level 1 Instruction Cache	64 KB, 4-way, 32 B
Memory Ports	2 Read, 1 Write
Branch Predictor Level 1/Level 2 Size History Register	2-Level Predictor 1/1024 entries 8 bits
Branch Misprediction Latency	6 cycles
Number of Cores	4
Shared Level 2 Unified Cache	2 MB, 8-way, 64 B
L1/L2/Memory Access Latencies	1/18/150 cycles
Thread Squash/Spawn/Sync Latencies	5/5/1 cycles

2.3. Compilation Infrastructure

Our compiler infrastructure is built on the Open64 Compiler [Open64 Developers 2001], an industrial-strength open-source compiler targeting Intel's Itanium Processor Family. We extended Open64 to extract speculative parallel threads from loops. The compiler estimates the parallel performance of each loop based on the cost of synchronization and the probability and cost of speculation failure, using loop nesting profile, edge frequency profile, and data dependence frequency profile. The compiler then chooses to parallelize a set of loops that maximize the overall program performance based on such estimations [Wang et al. 2005]. All binary codes are compiled with `-O3` optimization level. In addition, TLS-specific optimizations, such as interthread register, memory resident value communication, and reduction operations, are applied by the compiler to those parallelized loops [Zhai et al. 2002, 2004; Wang 2007].

2.4. Simulation Infrastructure

We build our simulation infrastructure based on a trace-driven, out-of-order superscalar processor simulator. The trace generation portion is based on the PIN instrumentation tool [Luk et al. 2005], and the architectural simulation portion is built on SimpleScalar [SimpleScalar LLC 2004] and augmented with 128 registers as defined in the Itanium ISA.

The trace generator instruments all instructions to extract information such as instruction addresses, registers used, memory addresses for memory instructions, opcodes, and so forth. The entire trace of the instruction stream is stored to disk files.

The simulator reads the trace files and translates the Itanium code bundles generated by the compiler into Alpha-like code. In addition to modeling register renaming, reorder buffer, branch prediction, instruction fetching, branching penalties, and memory hierarchy performance, we also extend the infrastructure to account for different aspects of speculative thread execution, including explicit synchronization through signal/wait, cost of thread commit/squash, and so forth. Table II shows the default architecture parameters of the TLS-enabled CMP. A set of variations of these parameters will be shown when the heterogeneous design space is explored in Section 3.1.

To estimate power consumption of the processors, the simulator integrates the Wattch [Brooks et al. 2000] model for core power consumption, the CACTI

[Shivakumar and Jouppi 2001][1] model for cache power consumption, and the Orion [Wang et al. 2002] model for interconnection power consumption. We assume the 70 nm technology.

2.5. Benchmark Workloads

Our target workloads are benchmark programs from the SPEC CPU2006 and CPU2000 suites written purely in C and C++. If a benchmark program appears in both suites, only one instance is evaluated. For example, we have evaluated 429.MCF from SPEC CPU2006 but omitted 181.MCF from SPEC CPU2000. For benchmark 175.VPR, we treated the PLACE and ROUTE input sets as two separate cases due to their distinct differences. They are referred to as VPR-P and VPR-R in later sections. All benchmarks are evaluated using the *ref* input sets. When there are multiple input sets, the first one is chosen for evaluation.

The following seven benchmark programs are not evaluated due to technical difficulties: 164.GZIP, 252.EON, and 255.VORTEX from SPEC CPU2000 and 471.OMNETPP, 483.XALANCBMK, 447.DEALII, and 450.SOPLEX from SPEC CPU2006. Nevertheless, we have evaluated our proposals on 25 different benchmark programs that exhibit diverse characteristics.

To reduce simulation time, we have adopted the SimPoint-based sampling technique [Perelman et al. 2006] with 100 million instructions per sample and up to 10 samples per benchmark. Up to 1 billion instructions can be simulated for each benchmark. With the sample size of 100 million instructions, the side effect of warming up is negligible. To ensure accuracy, the following sequence of operations is performed [Packirisamy 2007]: (1) The sequential binary is used to generate Basic Block Vectors, which are then used to select SimPoint samples. (2) The selected sample points are fed into the sequential trace generator, which creates traces for the corresponding points selected. (3) The parallel trace generator is augmented so that it selects the exact same code regions in the parallel binary as that in the sequential binary.

3. DESIGNING THE HETEROGENEOUS MULTICORE ARCHITECTURE

The TLS model is proposed to aggressively extract parallelism from sequential programs. In this section, we will look into the possibility of incorporating on-chip heterogeneous components to improve its energy efficiency. We start the investigation by building an idealistic execution environment to explore the design space. Based on the exploration, we present a feasible design solution and estimate its upper bound in energy efficiency improvement. A case study will show some interesting code behaviors and their performance and energy characteristics.

3.1. Design Space Exploration

Heterogeneous systems typically have a large design space. An idealistic execution environment would allow us to efficiently experiment with a large number of hardware configurations. We will measure the usage of each component and estimate the upper bound in energy efficiency.

3.1.1. Experiment Setup. To explore the three forms of heterogeneity, we construct an execution environment with the following computing and caching components: (1) one-, two-, four-, six-, and eight-issue cores with and without SMT support; (2) one-, two-, four-, and eight-way associative first-level caches of size 16KB, 32KB, 64KB, 128KB, and 256KB. Note that the second-level cache size is not a variable in our study. Architectural parameters of these cores are summarized in Table III. Our

[1]An old version, CACTI3, is used because it is the only available version integrated in the Wattch tool.

Table III. Architecture Parameters Scaled by the Issue Width

Core Type	1-Issue	2-Issue	4-Issue	6-Issue	8-Issue
Issue Width	1	2	4	6	8
Fetch Width	3	4	6	9	12
Commit Width	1	2	4	6	8
ROB Size	32	64	128	192	256
LSQ Size	24	32	64	96	128
Instr. Fetch Queue Size	8	16	32	64	64
Integer Units	1	2	4	6	8
Floating Point Units	1	3	4	5	6
Branch Predictor L2 Size	512	1024	1024	2048	2048

methodology in determining these numbers is to start with a reasonable configuration of the four-issue core (SimpleScalar default) and then scale them to other cores according to their respective issue widths. The L1 cache size varies by both the number of sets and the associativity; that is, the same cache size would vary from one-way to eight-way set-associative with a fixed 32-byte block size. Such variations apply to both data and instruction caches. The rest of the architectural parameters can be found in Table II.

During program execution, we refer to a sequence of dynamic instructions that is speculatively parallelized as a *parallel segment* and instructions sequentially executed between two parallel segments as a *sequential segment*. We limit the number of speculative threads in parallel segments to *four*, since previous work [Steffan et al. 2000] has shown limited scalability beyond four speculative threads for applications in the SPEC CPU benchmark suite. In a parallel segment, speculative threads can either be allocated to one SMT core (SMT mode) or to multiple non-SMT cores each executing one thread of execution (CMP mode); however, they cannot be spread across multiple cores each running multiple SMT threads (mixed mode). We omit the mixed-mode execution from our experiment to avoid the complexity involved in maintaining speculation states across two types of multithreading supports.

This execution environment is *idealistic* and we have made the following set of assumptions: (1) there can be as many cores and caches for each type as requested and all are connected through a bus, (2) powering on and off cores and caches incurs neither performance nor energy overhead, (3) cores and caches are immediately powered off after usage, (4) cache contents become available immediately after the cache is powered on (no warm-up cost), and (5) for each segment of execution, an “oracle” is able to correctly predict the most energy-efficient[2] processor configuration, measured in energy-delay-squared product (ED^{2}P). The following predictions are made: core issue width, L1 cache configuration (associativity and size), and to use SMT or CMP for parallel segment. In Section 5, we will evaluate our proposal with these assumptions removed.

3.1.2. Heterogeneous Component Usage Measurement. We first measure how often each configuration is used. Figure 2 shows, for each benchmark, the configuration usage breakdown measurements. The stack segment corresponds to the percentage of execution time in which a particular configuration is activated by the oracle, that is, achieving the highest energy efficiency. Configurations corresponding to less than 5% in all benchmarks are grouped in the `Others` category.

[2]The scope of optimality is limited to this idealistic execution environment. As various overheads are accounted in the real system (Section 5), the oracle prediction is no longer optimal.

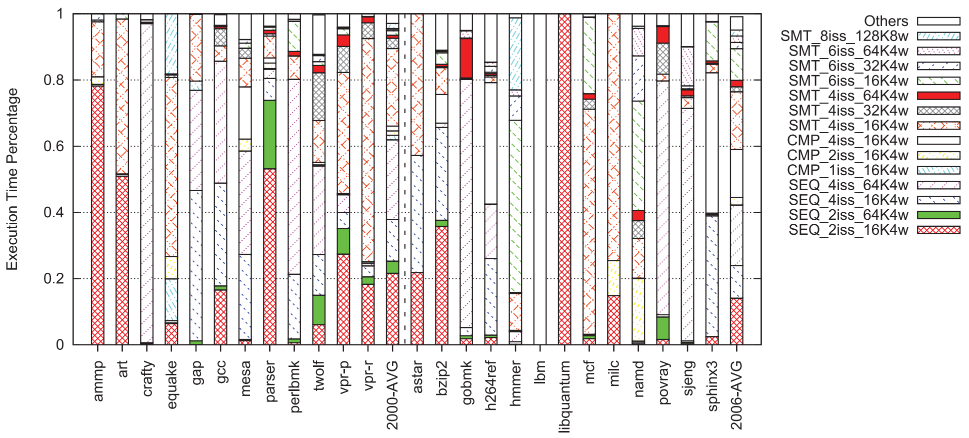


Fig. 2. Configuration usage breakdown measurements. The granularity is code segment.

Diverse configuration preferences are observed not only among different benchmarks but also within the same benchmark. We will highlight two cases to illustrate some of the key insights.

Benchmark LBM contains a high coverage loop (lbm.c:186) with a high degree of both thread-level and instruction-level parallelism. If speculative threads from this loop were allocated to a single core in the SMT mode, resource contention among threads would stifle the performance. This benchmark achieves the best ED^{2}P on four high-issue-width superscalar cores in the CMP mode (Guideline IV, Table I).

In LIBQUANTUM, the performance bottleneck is high L2 cache misses. For a high coverage loop (gates.c:89), the L2 miss rate is as high as 30%, and 85% of the processor cycles are stalled due to cache misses. CMP-based configurations with high-issue-width superscalar cores are clearly inefficient. SMT-based configurations can potentially tolerate cache miss latencies by running multiple threads in parallel. In this case, however, with only four threads, the SMT mode is unable to hide all L2 cache miss latencies. With respect to L1 cache size, the loop has very little reused cache blocks, and thus cannot benefit from a large cache. It turned out that the most energy-efficient configuration is a low-issue-width superscalar core with small L1 cache due to the low ILP (Guideline I, Table I).

It is worth pointing out that the four-way set associative cache is almost always the most energy-efficient choice. Reducing associativity is particularly harmful for speculative parallel threads, because these threads utilize cache ways to buffer speculative writes [Steffan et al. 2005]. Reducing cache associativity can increase the chances for speculative writes to overflow the cache and in turn cause speculation to fail.

3.1.3. Energy Efficiency Upper-Bound Estimation. Using the idealistic execution environment, we can estimate the upper bound of energy efficiency improvement compared to *homogeneous* designs.

We measure energy efficiency on a variety of SMT-based and CMP-based homogeneous systems that also vary in the same core issue widths and L1 cache configurations. Figure 3 shows the best of the SMT-based ones (bar `SMT-best`, four-issue core with 64K cache) and the CMP-based ones (bar `CMP-best`, four four-issue cores each with 64K cache). Bar `HET` stands for the heterogeneous execution environment. The normalization base is unmodified sequential code running on one four-issue *SMT* core with 64K four-way cache. Unless otherwise mentioned,this normalization base is used throughout the article.

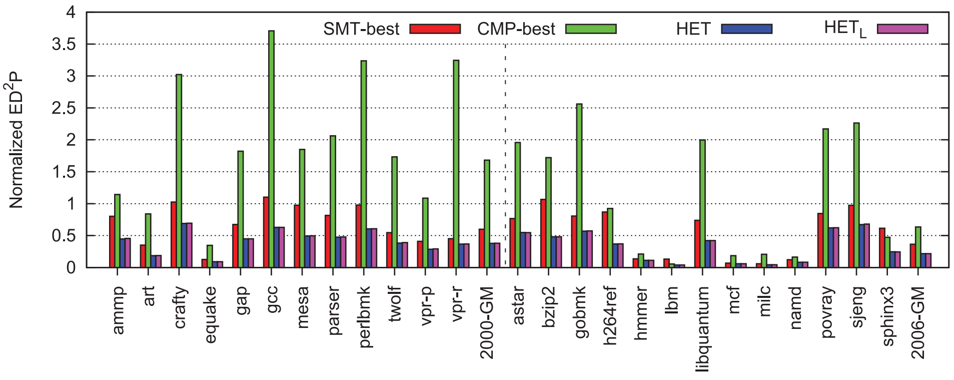


Fig. 3. Upper-bound estimation with respect to the best homogeneous systems. Bars are explained in Sections 3.1.3 and 3.1.4.

It is observed that even the best homogeneous systems sometimes degrade energy efficiency compared to the normalization base, for example, BZIP2 for SMT and GCC for CMP. The heterogeneous environment is always the most energy-efficient one. Across all the benchmarks, we have observed the upper bound of 38.8% and 72.1% in ED^{2}P reduction, compared to the best SMT-based and CMP-based homogeneous systems, respectively. The reduction compared to the normalization base is 71.7%.

3.1.4. Local vs. Global Decisions. Previous study [Sazeides et al. 2005] has shown that when performing dynamic optimizations for ED^{2}P, the best decision that minimizes ED^{2}P for a fraction of the total execution (local decision) may not lead to global reduction in ED^{2}P. Thus, dynamic optimizations that aim to reduce ED^{2}P face a fundamental difficulty: optimization decisions must be made with knowledge or information of the future. Fortunately, this has little impact in our study. The previous experiment HET makes globally optimal decisions (assuming the future is similar to the past). We have performed one extra experiment that makes locally optimal decisions (bar HET$_L$). We found negligible differences between these two experiments across all benchmarks (0.4%). Therefore, for the rest of this article, we allow the runtime system to make local decisions.

3.2. A Feasible Design Solution

Obviously a feasible solution could not integrate all the used configurations from the design space. Thus, it is crucial to identify a subset that (1) can capture most of the potential improvement and (2) is feasible to integrate. The component usage measurement (Figure 2) can give us many insights.

Across all the benchmarks, CMP-based configurations are activated for 11.4% of the total execution time, and SMT-based configurations are activated for 36.3%. Thus, both execution modes should be supported. Second, configurations with two-issue cores cover 22.0% of the total execution time, and configurations with four-issue cores cover 65.8%. Although six-issue cores have coverage of 9.1%, we found that the difference in energy efficiency is very small between four- and six-issue cores. Thus, we should only integrate two-issue and four-issue cores. Third, configurations with 16K, 32K and 64K four-way cache(s) represent 66.2%, 4.1%, and 26.8% of total execution time, respectively. Other cache configurations correspond to only 2.9%. Thus, we should integrate four-way associative caches, varying from 16K to 64K in size.

3.2.1. Proposed Heterogeneous Architecture. Figure 4 shows how these components are integrated. A *processing block* consists of one four-issue SMT core, one two-issue

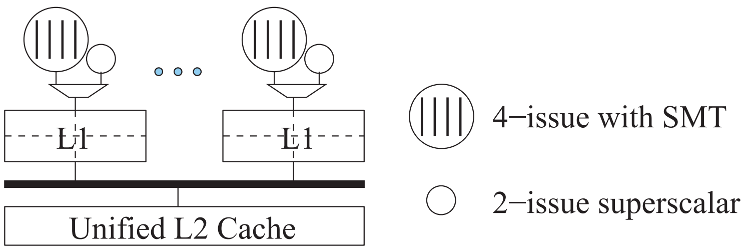


Fig. 4. Proposed heterogeneous multicore architecture for TLS. Each processing block consists of one four-issue SMT core, one two-issue non-SMT core, and one resizable L1 cache.

non-SMT core, and one 64K four-way L1 cache. The two cores share the L1 cache through a switch [Najaf-abadi et al. 2009]. The 64K L1 cache can be dynamically reconfigured to 16K and 32K in size. Because the four-way associativity needs to be maintained, we proposed to adjust the size by turning off part of the sets. A number of processing blocks are connected to the unified L2 cache. In this article, we limit the number of speculative threads to four, so each SMT core supports four threads of execution and four processing blocks are integrated. Given this degree of scalability, we employ a traditional snooping bus[3] as the interconnection media.

In this design, speculative threads can share one of the four-issue cores (SMT mode) or spread across multiple two-issue or four-issue cores, each running one thread (CMP mode). Sequential segments can choose between one of the two-issue and four-issue cores. Note that the four-issue SMT core will sometimes execute a single thread; this incurs some static power overhead that would otherwise not occur if the single thread executes on a non-SMT four-issue core. However, we choose not to integrate the latter so as to simplify the design and increase resource reusability so that more processing blocks may be integrated under a fixed transistor budget.

3.2.2. Understand Where Energy Efficiency Improvement Comes From. We now attempt to estimate the energy efficiency for this feasible design and also understand why it improves energy efficiency compared to homogeneous systems. We conduct our analysis by breaking the construction of our proposed architecture into incremental steps. In each step, we add one more form of heterogeneity to the best homogeneous system and measure its improvement to energy efficiency.

The most efficient homogeneous system is a four-issue SMT with 64K cache, shown as bar `Hom-best` in Figure 5. Here we assume that an SMT core consumes the same amount of energy as a non-SMT core when executing a single thread.

Multithreading Type: Bar $\texttt{Het}_{\texttt{Mt}}$ corresponds to the system that replicates `Hom-best` to support both SMT-mode and CMP-mode execution for speculative threads. Almost all the benchmarks are able to benefit from it significantly. ED^2P is reduced by 28.3% compared to `Hom-best`.

L1 Cache Size: Bar $\texttt{Het}_{\texttt{Mt+Ca}}$ builds on $\texttt{Het}_{\texttt{Mt}}$ and extends it with the ability to resize L1 cache between 64K and 16K. Plenty of benchmarks, such as AMMP, ART, GAP, ASTAR, BZIP2, and LIBQUANTUM, have shown noticeable improvement in energy efficiency. ED^2P is further reduced by 8.7%.

Core Issue Width: Bar $\texttt{Het}_{\texttt{Mt+Ca+Co}}$ builds on $\texttt{Het}_{\texttt{Mt+Ca}}$ and augments each four-issue core with a two-issue core. Benchmarks such as AMMP, ART, PARSER, BZIP2, and LIBQUANTUM show noticeable additional improvement in energy efficiency. ED^2P is further reduced by 3.8%.

[3]The interconnection media is orthogonal to this research and it is possible for our system to adapt to a more scalable choice.

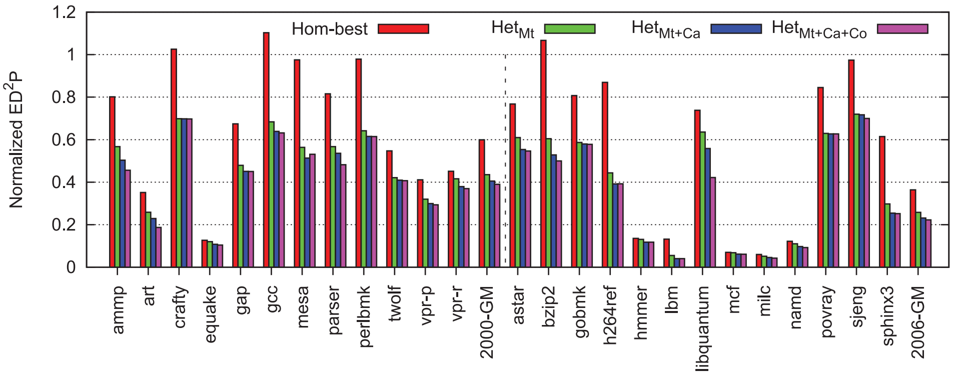


Fig. 5. Energy efficiency improvement comes from the three forms of heterogeneity: multithreading type, core issue width, and cache size. Bars are explained in Section 3.2.2. Also note that bar `Hom-best` is bar `SMT-best` from Figure 3.

Overall, our proposed heterogeneous architecture (bar $\texttt{Het}_{\text{Mt+Ca+Co}}$) can potentially improve energy efficiency by 37.0%, a close match to the upper bound of 38.8%, compared to the best homogeneous system. Thus, our selection of heterogeneous components is judicious.

3.2.3. Comparison with Single-Threaded Processors. We are also interested to know how much the proposed TLS heterogeneous architecture could improve over sequential[4] processors that execute the nonparallelized binary of the same benchmarks. In addition to the homogeneous sequential processor, we add another comparison baseline of a heterogeneous sequential processor that can adapt with the same choices in core issue width (two- and four-issue) and L1 cache capacity (16K four-way and 64K four-way).[5] For the latter, we construct an idealistic execution environment with the same set of assumptions used in Section 3.1.1 but executing nonparallelized binary code.

Figure 6 shows the comparison. Bar `SEQ-hom` is the sequential homogeneous baseline with one four-issue non-SMT core and 64K four-way L1 cache, which achieves the highest overall energy efficiency among other homogeneous configurations. Bar `SEQ-hete` and `TLS-hete` represent the heterogeneous sequential processor and heterogeneous TLS system, respectively.

Except for a few abnormalities where the thread-level parallelism is limited, `TLS-hete` performs significantly better than `SEQ-hete` in most benchmarks, leading to 46.8% improvement in ED^2P overall. Compared to `SEQ-hom`, `SEQ-hete` is quite similar in execution time and most of the 15.7% ED^2P improvement comes from energy reduction. As of `TLS-hete`, both execution time and energy consumption are reduced by 26.1% and 17.8%, respectively, leading to 55.1% ED^2P improvement compared to `SEQ-hom`.

It is shown that the heterogeneity we propose is beneficial to energy efficiency for both the single-threaded processor (15.7%, `SEQ-hete` vs. `SEQ-hom`) and the TLS system (38.8%, `TLS-hete` vs. `TLS-hom` in last section). Combining heterogeneous design and speculative parallelism, we can potentially improve performance (26.1%) and reduce energy consumption (17.8%) at the same time.

[4]The term "sequential" refers to a processor with single-threaded execution. It has nothing to do with whether the processor is in-order issue or not.

[5]Ideally, the heterogeneous sequential processor should have been constructed by profiling on all possible combinations of cores and caches and then deciding on a feasible architecture, which itself is a separate research project.

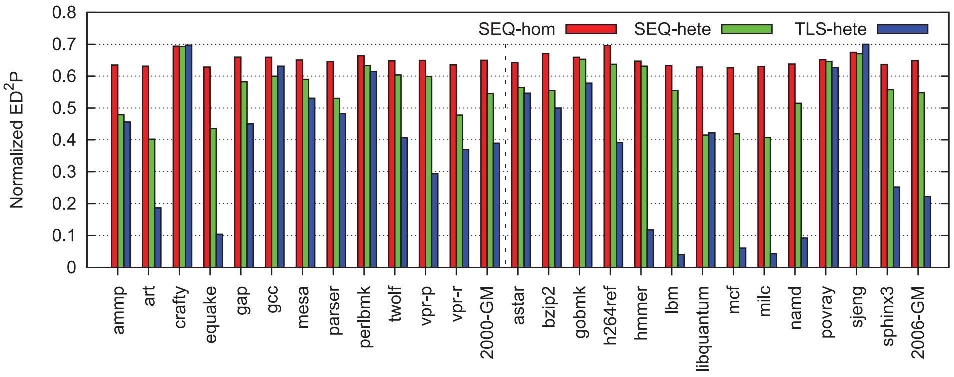


Fig. 6. Comparing with sequential processors (single-threaded). Bars are explained in Section 3.2.3. Also note that bar `TLS-hete` is the last bar in Figure 5.

3.3. Case Study

Many code segments exhibit interesting behaviors as they execute on different hardware configurations. In this section, we will focus on one such case and study these behaviors.

Figure 7(b) shows the execution time breakdown for executing this loop sequentially as well as in parallel on four-core CMP-based TLS systems. For our purposes, cycles are broken into six segments: `Busy`, cycles spent graduating non-TLS instructions; `dCache`, cycles stalled due to data cache misses; `ExeStall`, cycles stalled due to lack of instruction-level parallelism; `iFetch`, cycles stalled due to instruction fetch penalty; `Squash`, cycles wasted due to speculation failures; and `Others`, cycles spent on various TLS overheads, including thread spawning, committing, synchronization, and idling due to unbalanced workloads among threads. Bars represent different execution modes, core issue widths, and L1 cache capacities. In all configurations the L1 caches are four-way set-associative and 32 bytes in block size. Figure 7(c) shows their energy efficiency measured in ED^{2}P. All the bars are normalized to SEQ-4iss-64K (sequential code running on a four-issue core with 64K L1 cache), which is the sequential configuration that achieves the highest energy efficiency across all benchmarks.

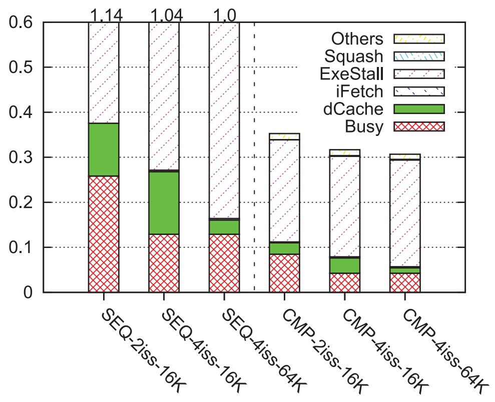
There is virtually no speculation failure in all TLS configurations as expected. Furthermore, the cycles spent on cache accesses are quite small in both sequential and TLS execution, and thus data cache performance is not a bottleneck for this loop. However, the execution stall (`ExeStall`) consumes a majority of the processor cycles, which indicates that individual iterations have low instruction-level parallelism.

Comparing the most aggressive sequential execution (SEQ-4iss-64K) and the least aggressive TLS execution (CMP-2iss-16K), we found that speculative parallelization reduces the execution time to 35.3%. When changing the two-issue cores to four-issue (CMP-2iss-16K vs. CMP-4iss-16K), the `Busy` segment is halved. However, the four-issue cores pose more stress on the L1 cache, resulting in a one-third increase in the `dCache` segment. Nevertheless, higher issue width improves performance by a noticeable margin. Although it is sensible to increase the L1 cache size (CMP-4iss-16K vs. CMP-4iss-64K), the benefit of bigger caches is offset by the increased execution stall, due to the limited instruction-level parallelism available in each loop iteration. The result is only slightly better than that of the smaller cache size. A similar trend is observed when moving from SEQ-2iss-16K to SEQ-4iss-64K.

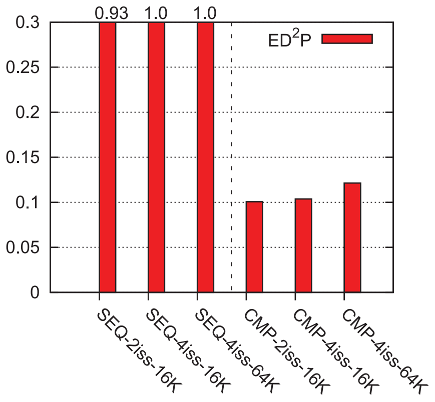
Higher issue width and larger cache capacity do not come free. When we trade off their performance benefit with the extra power consumption using the ED^{2}P metric, we find that configuration with a two-issue core and 16K L1 cache achieves the high-

```
for (k=0; k<npairi; ++k) {
   const int j = pairlisti[k];
   register const CompAtom *p_j = p_1 + j;
...
   int jfep_type = p_j->partition;
   BigReal lambda_pair = lambda_table_i[2*jfep_type];
   BigReal d_lambda_pair = lambda_table_i[2*jfep_type+1];
...
   reduction[pairVDWForceIndex_X] -= 2.0 * vdw_dir * p_ij_x;
   reduction[pairVDWForceIndex_Y] -= 2.0 * vdw_dir * p_ij_y;
   reduction[pairVDWForceIndex_Z] -= 2.0 * vdw_dir * p_ij_z;
...
   Force & fullf_j = fullf_1[j];
   register BigReal tmp_x = fullforce_r * p_ij_x;
   fullElectVirial_xx += tmp_x * p_ij_x;
   fullElectVirial_xy += tmp_x * p_ij_y;
   fullElectVirial_xz += tmp_x * p_ij_z;
   fullf_i.x += tmp_x;
   fullf_j.x -= tmp_x;
...
} // for pairlist
```

(a) NAMD code snippet in ComputeNonboundedBase2.h



(b) Execution time breakdown



(c) Energy efficiency

Fig. 7. Source code for a loop in benchmark NAMD and the execution time breakdown and energy efficiency as the loop runs on different configurations. Note that `Squash` and `Others` are always zero for sequential executions and the chopped portion in Figure 7(b) belongs to `ExeStall`.

est energy efficiency in both sequential and TLS execution modes. This is a strong case where simpler cores (two-issue) and smaller caches (16K) are significantly more efficient. Moreover, speculative parallelization further improves energy efficiency by 90% compared to the best sequential configuration for this loop (CMP-2iss-16K vs. SEQ-2iss-16K).

These types of program behaviors are not uncommon when the SPEC benchmark suites are speculatively parallelized. Thus, it is expected that heterogeneous design that matches the code behaviors can potentially improve energy efficiency by a significant margin.

4. HETEROGENEOUS SYSTEM IMPLEMENTATION

In the previous section, we have shown promising margins of energy efficiency improvement by integrating heterogeneous components into a multicore processor. To realize these potentials, two major hurdles must be overcome: switching overheads and resource allocation. We start the discussion by first presenting the runtime support.

4.1. Runtime Support

The runtime system maintains a hardware-based resource allocation table, similar to the decision table used by Luo et al. [2009] and Luo and Zhai [2012]. Indexed by the program counter address of the first instruction of each program segment, the resource allocation table keeps track of thread management decisions for each segment.

Before a segment of execution starts, this table is queried: if no entry is found, the configuration of one four-issue SMT core with a 64K L1 cache is activated as the *default* mode. This is also the most energy-efficient homogeneous configuration on average. When a segment of execution completes in this configuration, the runtime system predicts the optimal configuration and enters the decision into the resource allocation table.

4.2. Overhead Mitigation Mechanisms

A realistic heterogeneous system must take into account various runtime overheads associated with thread migration and resource reconfiguration. In particular, when threads operate at a fine granularity, these overheads can be significant.

4.2.1. Types of Runtime Overheads. Thread migration and resource reconfiguration can incur a performance penalty as well as extra energy consumption. In addition, an SMT core consumes more static power than a non-SMT core when executing a single thread.

Startup overhead. When the runtime system migrates a thread to a core that was previously powered off, execution cannot resume immediately. We model a 3,000-cycle startup cost for the four-issue core and an 800-cycle cost for the two-issue core [Isci and Martonosi 2003].[6] During this period, static power is being consumed.

Cache reconfiguration overhead. Powering off a cache or part of a cache has additional costs. When a cache is powered off or resized to a smaller cache size (i.e., lower number of sets), dirty lines must be written back to the next-level cache, and some contents are discarded. When resizing to a bigger cache size (i.e., higher number of sets), the same operations must be performed, since some tag bits are shifted to the index bits after increasing the number of sets and the mapping of an address to a cache set is changed.

SMT overhead. Our SMT architecture is implemented by extending a superscalar core of the same issue width and replicating the renaming unit and the register file. When an SMT core is executing a single thread, the replicated structures are not used but continue to consume static power. Thus, the cores with SMT support could incur additional overhead in energy consumption.

4.2.2. Mitigating the Impact of Overheads. Frequent thread migration and resource reconfiguration are clearly detrimental. The central idea behind the various mitigation heuristics is to reduce the frequency of such operations while still exploring the benefit of having heterogeneous components.

Coalescing requests for small segments. When a segment of execution is short, migrating threads or reconfiguring resources would not justify the cost. In fact, we have observed a number of situations where small segments with different configuration preferences interleave. This creates tremendous overheads if reconfiguration is permitted whenever it is requested (Guideline V, Table I).

Ideally, we should prohibit thread migration and resource reconfiguration for all segments that execute less than a certain number of cycles. But this would require

[6]These cycle numbers are from empirical analysis guided by the principles learned from the given reference. The rest of the results are insensitive to these cycle numbers because the percentage of execution time that stalled on startup overhead is as small as 0.1% (Section 5.6).

knowing the cycle number before actually executing the invocation of a segment. Another problem is that although each invocation is small in cycle, a large number of consecutive invocations can together create a reasonable demand for thread migration or resource reconfiguration. Checking only the individual invocations would miss many opportunities.

To cope with these problems, we propose two heuristics: *accumulation* and *approximation*. The runtime system postpones thread migration and resource reconfiguration until the demand for such an operation has *accumulated* to a certain threshold. The accumulation is the cycle sum of the invocations of segments whose resource demands are the same but not met. The threshold should be related to the cost of the operation. Since cache reconfiguration exhibits additional costs, a higher threshold is set. Before the threshold is reached, reconfiguration requests are denied and the runtime system continues the execution with the current configuration. Meanwhile, the runtime system attempts to *approximate* the desired configuration with the components that are currently powered on. For example, the current configuration is four-issue 16K and an incoming sequential segment demands two-issue 16K. If no two-issue core is powered on in the system, the runtime system can schedule this segment on a configuration of a four-issue core with a 16K cache.

Postponing component power-off. Cores and caches should not be immediately powered off when they are no longer needed. There are many cases in which a sizeable parallel segment frequently interleaves with a small sequential segment. This usually happens when an inner loop in a loop nest is parallelized. In every iteration of its immediate parent (outer) loop, a sizeable parallel segment corresponds to the inner loop invocation, and a small sequential segment corresponds to instructions around the inner loop, such as those updating the outer loop's variables, and so forth. As the parallel segment finishes, many components are not needed by the imminent sequential segment. Turning off these components immediately is problematic. Postponing the powering-off using a deterministic delay can solve this problem since the sequential segment is short and does not exhibit much variance. We are able to find an empirical threshold that works for most cases.

Mitigation for SMT overhead. Most of the SMT overheads come from the replicated register files. We propose to power-gate the replicated register files, that is, powering them on and off as needed. A 500-cycle delay is accounted when powering them on. Similar mitigation mechanisms (i.e., accumulation, approximation, and postponing powering-off) are applied.

4.3. Resource Allocation Schemes

The runtime support needs to decide, for each segment of execution, what resources should be allocated to execute the threads. Up to this point, an oracle makes this decision by first profiling each segment on all possible hardware configurations and then choosing the best one [Kumar et al. 2003]. However, this is not realistic at runtime given that the number of possible configurations can be quite large, even with a few degrees of freedom.

In this section, we propose a realistic resource allocation scheme that utilizes hardware performance counters to monitor program execution and predict the most energy-efficient configuration for each segment. The scheme profiles each segment of execution once when it is first invoked. The profiling phase allocates all threads to the *default* configuration that has one four-issue SMT core with a 64K L1 cache. Based on the profiling run, the optimal configuration is predicted and saved in the resource allocation table until its entry is reset. This approach naturally adapts to different input sets and can potentially adapt to phase change behaviors if the whole table is reset at the end of each phase. A configuration contains the following information: core issue width (four-issue or two-issue), L1 cache size (64K or 16K), and the multithreading type (CMP or SMT, for parallel segment only).

4.3.1. Determining Core Issue Width. The instruction per cycle (IPC) metric is often used to measure the performance of a processor, but it sometimes can be a poor indicator for whether the four-issue core is efficiently utilized.

For program segments with a large number of long-latency operations, even though having low IPCs, they may still require a large reorder buffer (ROB) to hold the instructions and exploit instruction-level parallelism to sustain these IPCs. Normally, a four-issue core would have a larger ROB than the two-issue core. Thus, these program segments may be more energy efficient on the four-issue core because of the extra ROB entries, even though the IPC statistics appear to be low.

We take the following approach to accommodate this effect. When profiling a segment on the four-issue core, we record the percentage of instructions that are issued when they are farther from the ROB head than the size of the two-issue core ROB. For example, in our design the four-issue core has an ROB twice the size of that in the two-issue core (Table III). Thus, the number of instructions that are issued from the second half of the ROB is counted. We denote the fraction of this count over the total number of issued instructions as the *deep-issue rate*. Thus, either a high deep-issue rate or a high IPC indicates that the segment can benefit from a more powerful core to exploit ILP, and thus the four-issue cores should be selected; otherwise, the two-issue cores are selected.

4.3.2. Determining L1 Cache Size. The *reuse rate* of cache blocks is used as an indicator of whether a cache is efficiently utilized. It is calculated as the fraction of the number of unique cache blocks that are accessed more than once over the total cache accesses during the profiling period.

This mechanism can be implemented with moderate hardware cost: two bits per cache block and two counters per cache. The two bits are used to indicate whether an individual cache block is accessed more than once and whether this block is already considered in counting the number of unique blocks. The two counters are used to count the number of reused blocks and the number of total cache accesses. Given the 32-byte block size, the space overhead is less than 1%.

We only need to find out whether 64KB cache is efficiently utilized; thus, we employ an empirical threshold for reuse rate. If a 64KB cache does not meet the threshold during the profiling period, this segment will use a 32KB cache. We acknowledge that this single threshold is a simplification of a bigger problem that determines the efficiency of cache usage. Nevertheless, this simple solution works well for our purpose and yields quite moderate architecture cost.

4.3.3. Determining Multithreading Type. SMT and CMP exhibit different power and performance characteristics in supporting speculative threads. To determine the optimal execution mode, we propose to measure the level of contention on the default four-issue SMT core. If the threads are consistently stalled at the issue stage because the required resources or function units are occupied by other threads, the CMP mode will be chosen so that the computation can be spread across multiple cores. Otherwise, the SMT mode is chosen. Eyerman and Eeckhout [2009] have proposed ways to obtain accurate cycle breakdown for threads executing under the SMT mode, which can be adopted to further improve the measurement of thread contentions.

New hardware counters must be integrated to the core and L1 cache to provide the required statistics. Note that we profile and make decisions only once for each segment. For instance, for the same segment, we start with the bigger cache and possibly convert

Table IV. Speculative Thread Characteristics in SPEC CPU2000 and CPU2006 Benchmarks

CPU2000	Thread Size	Count	Coverage	CPU2006	Thread Size	Count	Coverage
ammp	363.2	23.7	99%	astar	276.6	799.5	71%
art	48.6	4,323.4	100%	bzip2	162.7	7.0	46%
crafty	1,764.2	1.9	13%	gobmk	919.9	3.3	20%
equake	113.9	6.5	93%	h264ref	175.5	23.6	81%
gap	654.9	101.2	27%	hmmer	221.6	148.2	96%
gcc	36.0	32.1	83%	lbm	407.4	122,606.1	100%
mesa	103.7	2.5	60%	libquantum	17.3	795,282.1	100%
parser	209.0	4.3	75%	mcf	62.6	16.6	97%
perlbmk	140.5	4.2	25%	milc	235.2	62,186.9	85%
twolf	261.4	1.9	46%	namd	102.0	59.1	99%
vpr-p	219.7	4.9	56%	povray	746.3	2.4	64%
vpr-r	62.9	5.9	90%	sjeng	162.5	4.8	40%
Average	331.5	376.0	64%	sphinx3	302.2	32.4	98%
				Average	291.7	7,5474.8	77%

Thread Size: average number of dynamic instructions per thread.
Count: average number of threads per parallel segment.
Coverage: the fraction of time executing instructions in parallel segments.

it to a smaller cache; but the opposite transition never happens. It is the same with core size and execution mode. Potentially, we could reset the resource allocation table so that all segments are profiled again to cope with phase change behaviors in the program.

5. SYSTEM EVALUATIONS

In this section, we will first evaluate the impact of runtime overheads and the effectiveness of mitigation mechanisms. Then, we will compare the oracle and the realistic resource allocation schemes. Last, the energy efficiency of our proposed system will be contrasted with different baselines.

5.1. Benchmark Characteristics

Table IV summarizes some important characteristics of speculative threads for SPEC CPU2000 and CPU2006 benchmarks. We find that the average thread size, defined as the average number of dynamic instructions, is only about 330 for CPU2000 and 290 for CPU2006; the average number of threads in a parallel segment is less than 150 in a majority of the benchmarks. Moreover, these speculative threads collectively cover 64% and 73% of total execution in CPU2000 and CPU2006, respectively. Thus, to realize the energy efficiency potentials, the runtime system must be able to manage threads at a *fine* granularity and cope with the associated overheads.

5.2. Evaluation of Runtime Overheads

There are three types of runtime overheads, namely, startup overhead, cache reconfiguration overhead, and SMT overhead. We will measure their impacts separately. Note that the oracle resource allocation scheme is assumed in this subsection.

Measuring cache reconfiguration overhead. In Figure 8, segment PT stands for the energy efficiency assuming no overhead. Segment $\mathtt{OH}_1$ shows the degradation in energy efficiency when the cache reconfiguration overhead is first taken into account. This type of overhead is severely manifested in almost every benchmark.

Take benchmark SJENG as an example. A speculatively parallelized loop, located at line 227 in source file neval.c, is most energy efficient with a 16K cache configuration. However, its neighboring sequential segment achieves optimal energy efficiency with

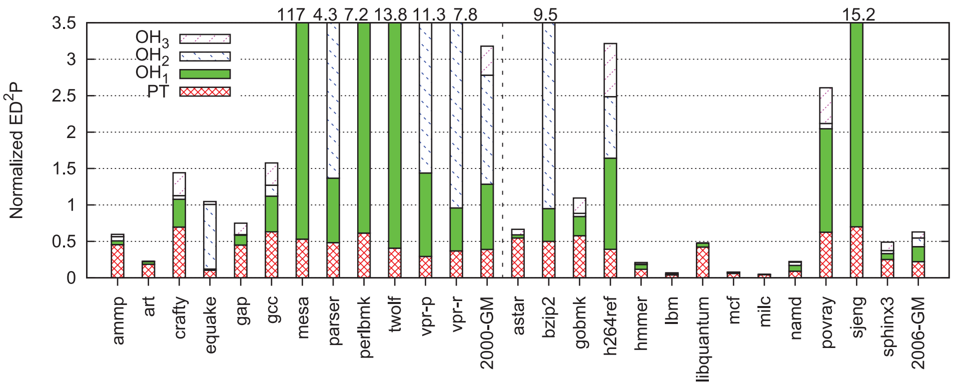


Fig. 8. Measuring the three types of runtime overheads impacting the proposed heterogeneous system. Segments are explained in Section 5.2. Also note that PT is the last bar in Figures 5 and 6. To improve readability, the y-axis is limited to 3.5. Detailed breakdown can be found in Table V.

Table V. Impact of the Three Types of Overheads: Detailed Breakdown

CPU2000	OH_1	OH_2	OH_3	CPU2006	OH_1	OH_2	OH_3
ammp	8.5%	9.9%	5.2%	astar	6.5%	0.1%	11.3%
art	16.7%	0.5%	0.4%	bzip2	4.7%	75.0%	15.0%
crafty	26.4%	3.5%	21.7%	gobmk	24.1%	3.9%	19.2%
equake	1.6%	84.6%	3.9%	h264ref	38.9%	26.2%	22.7%
gap	18.0%	1.2%	20.9%	hmmer	30.7%	3.5%	9.8%
gcc	31.1%	9.6%	19.3%	lbm	15.7%	0.0%	25.5%
mesa	29.7%	51.8%	18.0%	libquantum	11.6%	0.0%	0.0%
parser	20.6%	53.9%	14.4%	mcf	8.0%	13.4%	1.3%
perlbmk	71.3%	0.3%	19.9%	milc	9.8%	0.0%	0.0%
twolf	36.4%	52.7%	7.9%	namd	34.8%	20.2%	3.8%
vpr-p	10.2%	77.8%	9.4%	povray	54.4%	2.8%	18.8%
vpr-r	7.6%	81.9%	5.8%	sjeng	77.0%	1.0%	17.4%
				sphinx3	16.6%	8.6%	23.3%

Percentage numbers are with respect to the sum of all four components in Figure 8.

a 64K cache configuration. These two segments are nested in an outer loop that has a large iteration count. Thus, cache resizing occurs frequently. Since cache contents are lost or partially lost during resizing, this results in severe performance degradation and increased energy consumption.

Overall, the heterogeneous system spends 47.9% more energy and suffers 22.9% performance loss, that is, a 148.9% degradation in ED^2P due to cache reconfiguration overheads.

Measuring startup overhead. Segment OH_2 corresponds to the additional degradation as the startup overhead is considered. A number of benchmarks, such as EQUAKE, MESA, PARSER, TWOLF, VPR-P, VPR-R, BZIP2, and H264REF, are greatly affected by the startup overhead. Overall, it leads to 160.7% additional degradation on top of OH_1. Combining these two sources of overheads, energy efficiency is degraded by 309.6%.

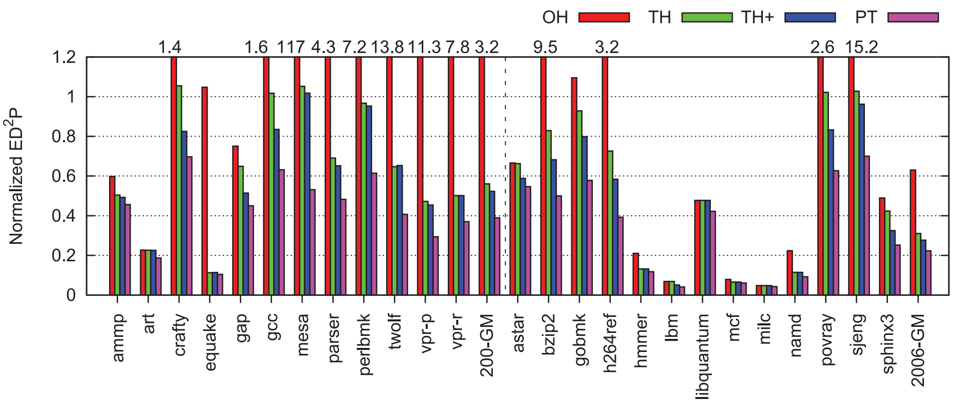


Fig. 9. Measuring the effectiveness of the mitigation mechanisms. Bars are explained in Section 5.3. Also note that OH is OH_3 in Figure 8 and PT is the same as in Figure 8.

Measuring SMT overhead. Segment OH_3 stands for the additional degradation as the SMT overhead is further accounted. About half of the benchmarks, where the SMT core(s) are constantly used to run a single thread, suffer significantly: CRAFTY, GAP, GCC, MESA, PARSER, PERLBMK, TWOLF, VPR-P, VPR-R, BZIP2, GOBMK, H264REF, POVRAY, SJENG, and SPHINX3. Overall, it leads to another 61.5% degradation in energy efficiency on top of OH_2. When the three types of overheads are combined, energy efficiency is degraded by as much as 370.7% compared to the heterogeneous system assuming none of these overheads.

When compared with the homogeneous system, the 37.0% potential improvement (estimated in Section 3.2.2) in energy efficiency is turned into a 196.4% degradation.

5.3. Evaluation of Mitigation Mechanisms

These runtime overheads clearly reverse the benefit of the proposed heterogeneous system. While they are inherent because of the heterogeneous nature of the system, we could mitigate their impacts by reducing the frequency of causal operations.

In Figure 9, bar OH stands for the degradation caused by all three types of overheads. When we apply the mitigation mechanisms for thread migration and cache reconfiguration (bar TH), the overheads are greatly mitigated for almost every benchmark.

Let's re-examine the case of the loop from SJENG, located at line 227 of source file neval.c. Although the neighboring sequential segment can potentially benefit from a larger cache, it is very small and will never reach the accumulation threshold. The mitigation mechanisms correctly reject the cache reconfiguration request from each invocation of the small sequential segment and protect the cache from constantly being resized. Thus, the overhead impact is significantly reduced.

Further applying the mechanisms to power off SMT registers (bar TH+) has additional benefits in reducing the overhead impact. A number of benchmarks are able to take this advantage, such as CRAFTY, GAP, GCC, ASTAR, BZIP2, GOBMK, H264REF, POVRAY, SJENG, and SPHINX3.

Overall, these mitigation mechanisms reduce the impact of overheads from 370.7% (OH vs. PT) to 29.0% (TH+ vs. PT), a reduction of as much as 92.2%. Though there may still be room for further improvement, we leave the investigations to future work.

5.4. Evaluation of Resource Allocation Schemes

The energy efficiency of the heterogeneous system using the realistic resource allocation schemes is shown as bar RE in Figure 10. It is able to realize most of the improvement

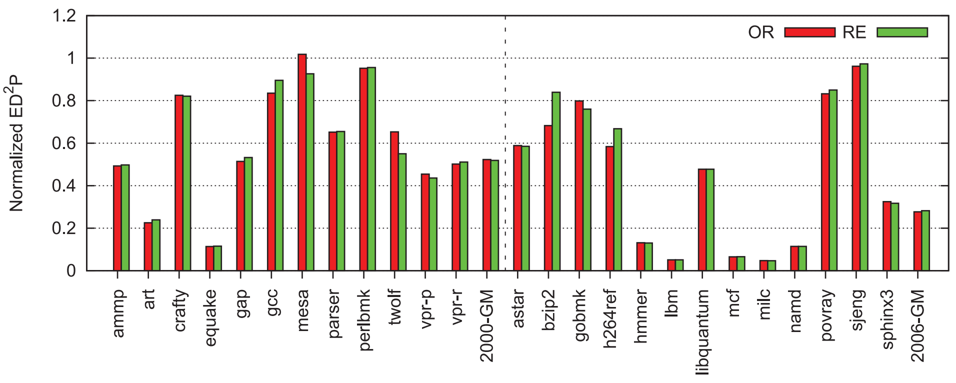


Fig. 10. Comparing the energy efficiency of the oracle and the realistic resource allocation schemes. Bars are explained in Section 5.4. Also note that OR is TH+ in Figure 9.

demonstrated by the oracle allocation (bar OR). Only a few benchmarks exhibit noticeable degradation compared to the oracle allocation, such as GCC, BZIP2, H264REF, and POVRAY, due to the deviations in configuration selections.

It is worth pointing out that the so-called oracle allocation is no longer optimal once runtime overheads are accounted. The cost of configuration change must be taken into consideration. In addition to the mitigation mechanisms, the prediction-based realistic allocation could further reduce the number of configuration changes by enforcing one configuration per segment. This explains why RE is sometimes better than OR, for example in MESA, TWOLF, and GOBMK.

Overall, the difference between OR and RE is only 0.7%. Thus, we conclude that our proposed resource allocation schemes are making the right decisions in choosing the configuration for each segment of execution.

5.5. Contrasting the Heterogeneous System with Various Baselines

In this section we will contrast the energy efficiency of the proposed heterogeneous implementation (bar Het-TLS) with the following three baselines: (1) the sequential processor that achieves the highest energy efficiency among other sequential configurations (bar Hom-Seq), (2) the heterogeneous sequential processor that has the same types of heterogeneity as in the proposed heterogeneous TLS implementation (bar Het-Seq), and (3) the homogeneous TLS configuration that achieves the highest energy efficiency (bar Hom-TLS). Their energy efficiency corresponds to the bars in Figure 11.

5.5.1. Hom-Seq vs. Normalization Base. Our normalization base is the unmodified sequential code running on a four-issue SMT core with a 64K four-way L1 cache. The only disparity between Hom-Seq and the normalization base is whether the core has SMT support.

Since their execution time and dynamic power consumption are virtually the same, the difference between the Hom-Seq bar and 1 (normalization base) reveals the magnitude of the extra static power consumption when an SMT core is used to execute a single thread. Across all benchmarks, the degradation in energy efficiency due to SMT support is 54.1%.

5.5.2. Het-Seq vs. Hom-Seq. In Section 3.2.3, we estimated the upper bound of energy efficiency improvement when heterogeneity is introduced to the sequential processor. In this section, we have implemented such a heterogeneous sequential processor: A four-issue core is augmented with a two-issue core and they are connected through a switch to an L1 cache that can be resized between 64K four-way and 16K four-way by

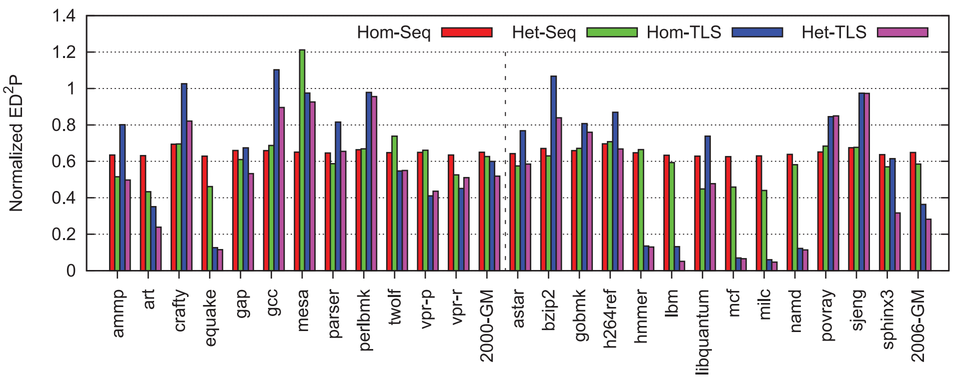


Fig. 11. Comparing the energy efficiency of the proposed heterogeneous system with various baselines. Bars are explained in Section 5.5. Also note that `Hom-Seq` is `SEQ-hom` in Figure 6, `Hom-TLS` is `Hom-best` in Figure 5, and `Het-TLS` is `RE` in Figure 10.

powering on and off some of the cache sets. This processor is also equipped with the same overhead mitigation mechanisms.

Noticeable improvement in energy efficiency is observed in 14 benchmarks, such as ART and MILC. In other benchmarks, the improvement is limited by the overheads, even with mitigation mechanisms. Across all benchmarks, the heterogeneous sequential processor is 6.8% better than the homogeneous sequential processor in terms of ED^{2}P.

5.5.3. Het-TLS vs. Het-Seq. Our findings about heterogeneous sequential processors are in line with Kumar et al. [2003]. Introducing speculative parallelism further magnifies the benefit of heterogeneous design. This is demonstrated by the 37.5% improvement in ED^{2}P from a heterogeneous sequential processor (`Het-Seq`) to a heterogeneous multicore processor that executes speculative parallelized code (`Het-TLS`).

Conclusion: Results from the last two sections reveal the separate contributions to energy efficiency improvement from heterogeneity (6.8%) and speculative parallelism (37.5%). Without speculatively parallelizing sequential programs, the impact of heterogeneous design can be very limited.

5.5.4. Het-TLS vs. Hom-TLS. Our proposed heterogeneous TLS system outperforms the most energy-efficient homogeneous TLS configuration in 21 out of 25 benchmarks, and by a large margin in AMMP, ART, CRAFTY, GAP, GCC, PARSER, ASTAR, BZIP2, H264REF, and SPHINX3. The only noticeable degradation is in VPR-P and VPR-R due to the overheads we have identified in Section 4.2 and evaluated in Section 5.2.

For SPEC CPU2000 benchmarks, `Het-TLS` reduces energy consumption by 21.7% with 4.9% performance loss,[7] which is a 13.4% improvement in energy efficiency, measured by ED^{2}P. For SPEC CPU2006 benchmarks, `Het-TLS` improves performance by 4.5% and saves energy by 15.3%, which is a 22.4% improvement in energy efficiency. Across all benchmarks, the energy efficiency is improved by 18.2% compared to the homogeneous TLS baseline.

Conclusion: The heterogeneous design is essential to further boost energy efficiency over prior TLS implementations by a significant portion.

5.5.5. Het-TLS vs. Hom-Seq. Last, the proposed heterogeneous TLS system is compared with the most energy-efficient sequential configuration. For a few benchmarks that

[7]The heterogeneous TLS system achieves slightly less performance but better overall energy efficiency compared to the much more aggressive homogeneous TLS configuration—a common practice in performance and energy tradeoff.

Table VI. Contrasting with the Homogeneous Sequential Processor (Hom-Seq)

	Heterogeneous sequential (Het-Seq)	Homogeneous TLS (Hom-TLS)	Heterogeneous TLS (Het-TLS)
Execution time reduction	–4.8%	22.3%	22.2%
Performance (speedup)	0.954X	1.288X	1.286X
Energy reduction	15.2%	–18.1%	3.6%
EDP improvement	11.1%	8.3%	25.1%
ED^2P improvement	6.8%	28.8%	41.8%

have large sequential portions with high instruction-level parallelism, Het-TLS is less efficient than Hom-Seq, such as in CRAFTY, GCC, POVRAY, SJENG, and so forth. Nevertheless, Het-TLS is significantly better than Hom-Seq in 15 out of 25 benchmarks.

For SPEC CPU2000 benchmarks, Het-TLS increases energy consumption by 1.1% but improves performance by 12.5%, which is a 20.2% improvement in energy efficiency, measured by ED^2P. For SPEC CPU2006 benchmarks, Het-TLS improves performance by 45.5% and consumes 7.9% less energy, which is a 56.5% improvement in energy efficiency. Across all benchmarks, the energy efficiency is improved by 41.8% compared to the homogeneous sequential baseline.

Conclusion: Through heterogeneous design and speculative parallelism, performance is improved over sequential execution in an energy-efficient way.

5.5.6. Summary. We summarize the comparisons among the above baselines using Table VI, which contrasts execution time, energy consumption, energy-delay product (EDP), and ED^2P against the homogeneous sequential processor.

It is observed that overall, the heterogeneous sequential processor reduces energy at the cost of lower performance, the homogeneous TLS system improves performance at the cost of higher energy, and only the proposed heterogeneous TLS system can both improve performance and reduce energy consumption. In addition, we show the execution time and energy consumption for each benchmark in Figure 12.

5.6. Measurement of Configuration Usage Breakdown

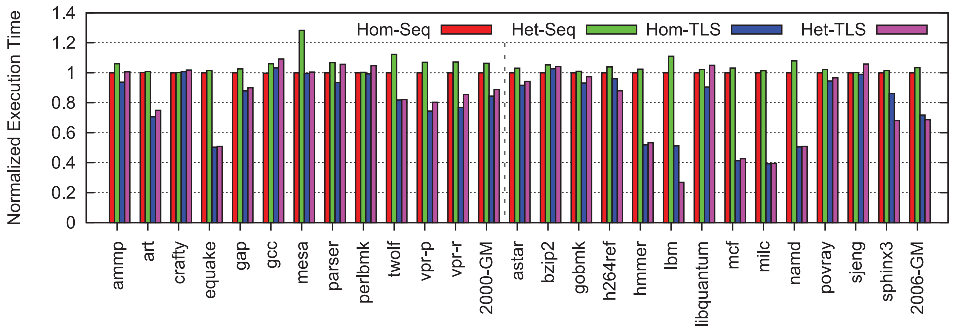
We have measured, in the proposed heterogeneous system, the fraction of execution time in which each configuration is activated. Figure 13 reveals diverse use of configurations. In SPEC CPU2000 benchmarks, CMP-based configurations are rarely used due to their low level of thread-level parallelism. In SPEC CPU2006 benchmarks, CMP-based configurations are activated for 10% of total execution time, because the higher level of thread-level parallelism available could justify the cost of using CMP. Across all the benchmarks, the four-issue and two-issue cores are used for 79% and 21% of total execution time, respectively, and the 64K four-way and 16K four-way cache(s) are used for 32% and 67% of total execution time, respectively.

Furthermore, thread migration and/or cache resizing have taken place in the same program at runtime. For example, in ASTAR, both four-issue-core and two-issue-core configurations are used significantly; this is evidence for thread migration. In H264REF, both 64K and 16K caches are activated for a large percentage; this is evidence for cache resizing. Similar cases can be found in a number of benchmarks.

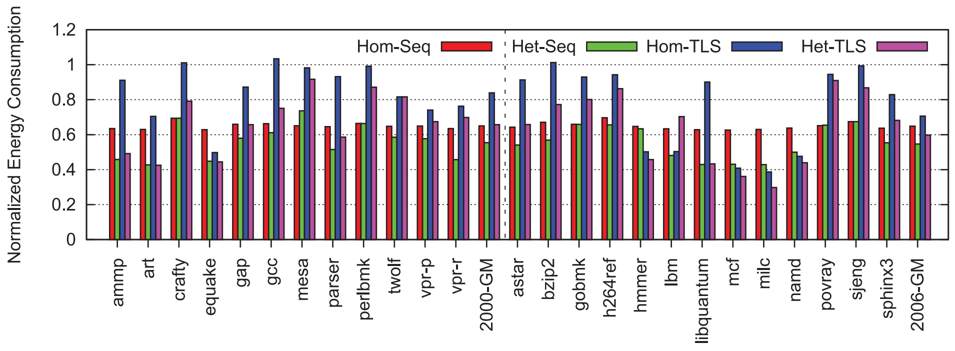
It is also worth pointing out that the cycles that are stalled due to powering on a component are minimal (0.1%). Thus, the mitigation mechanisms are very effective in reducing the startup overheads.

6. RELATED WORK

This work is the extension of a previous publication [Luo et al. 2010]. It is closely related to the following areas of computer architecture research: TLS, dynamic optimization,



(a) Execution time



(b) Energy consumption

Fig. 12. Execution time and energy consumption comparisons. Bars are defined the same as in Figure 11.

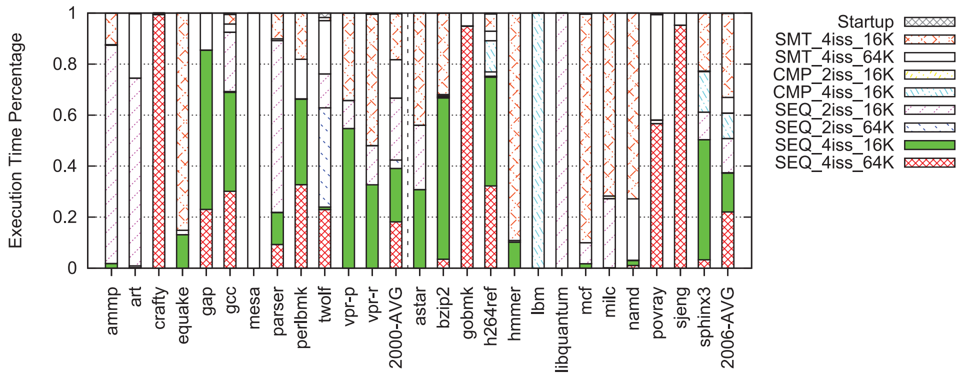


Fig. 13. Execution time breakdown of the programs running on the proposed heterogeneous system. Stack segments correspond to the percentage of execution time a certain hardware configuration is activated or the stalled cycles due to powering on a component (segment `Startup`).

energy-efficient computation, and heterogeneous multi-core design and implementation. In this section, we briefly outline related works in these areas.

A number of prior works have addressed energy efficiency issues for TLS execution. Packirisamy et al. [2008] compared and contrasted the energy efficiency of SMT and CMP in terms of supporting TLS workload under the same die area constraints. They examined how the interactions between speculative threads affect their energy efficiency in SMT-mode and CMP-mode execution. Renau et al. [2005] proposed various energy-saving optimizations for TLS execution. Tuck et al. [2007] built a task-criticality graph and used it for scheduling speculative threads based on their critical levels. Sarangi et al. [2005] introduced *ReSlice* that could significantly reduce the cost of value mis-speculation and used it to improve both performance and energy efficiency for TLS systems. Our work differs from the above in that (1) we proposed an asymmetric multicore system tuned for TLS execution, and (2) our energy efficiency improvement mainly came from successfully matching program segments with computation and caching components on which they are most energy efficient. Nevertheless, it is worth pointing out that many of the techniques proposed by these prior works are orthogonal and can potentially complement our system to further improve energy efficiency.

Managing speculative threads at runtime can reduce the unpredictability of speculative threads. Luo et al. [2009, 2012] proposed dynamic dispatching mechanisms to enhance TLS performance. Novel hardware performance counters are introduced to monitor thread execution and enable quantitative evaluation of speculative threads. Based on such evaluations, the runtime tuning system can (re-)decide how the speculative threads are extracted to improve performance. This work is based on a homogeneous CMP system and does not consider energy consumptions. However, these techniques can potentially be integrated into our heterogeneous system to extract more energy-efficient speculative threads from sequential applications.

A large number of proposals have discussed dynamic tuning for improving energy efficiency of microprocessors. Wu et al. [2005a, 2005b] presented the runtime optimizer framework that uses DVFS to manage energy versus performance tradeoffs. This framework is based on dynamic instrumentation: initial test instructions are inserted to hot code regions to collect the execution profile; then mode set (reset) instructions are inserted at the entry (exit) point of candidate code segments. Bhattacharjee et al. [2009] proposed a thread criticality predictor and then applied dynamic optimization and DVFS based on the criticality. While DVFS is not evaluated in this article, it is certainly orthogonal to our approach.

As more and more transistors are integrated onto a single die, efficiently utilizing on-chip resources becomes increasingly challenging. The work from Kumar et al. [2006], Chakraborty et al. [2006], and Becchi and Crowley [2006] studied thread migration for better resource utilization. However, their proposed schemes worked at coarse granularity and may not support speculative threads. Our work focused on speculative threads and tackled fine-grained thread management overheads.

Heterogeneous systems with same-ISA heterogeneous cores are proposed by Kumar et al. to improve energy efficiency for sequential programs [2003] as well as multiprogramming workloads [2004]. They presented policies for matching core sizes with application characteristics. Our work is different because (1) we studied speculative threads that have unique sharing and interference patterns; (2) our threads are managed at a much finer granularity level, and thus our runtime system must throttle the invocation of thread management operations; and (3) we have shown the benefit of incorporating heterogeneous cache components, which is not considered by Kumar et al. Suleman et al. [2009] proposed an Asymmetric Chip Multiprocessor to speed up lock-based critical section execution in OpenMP applications. Our work differs in that our workloads do not have explicit lock-based critical sections and the runtime deals with resource allocation for *all* program segments. Yang et al. [2002] proposed first-level cache designs that can be resized during program execution. Considerable energy-delay product reduction has been observed by having the ability of resizing both sets and associativities. Our work has shown that in the context of speculative threads that utilize cache ways as write buffers, resizing by sets is always a more energy-efficient choice. In addition, Patsilaras et al. [2012] studied workloads with high memory-level parallelism that resulted in a heterogeneous multicore system very similar to our design. It shows that the types of heterogeneous components we advocate have applications not only for speculative thread-level parallelism but also for memory-level parallelism.

In the effort to implement heterogeneous processors, FabScalar [Choudhary et al. 2011, 2012] presented a toolset to automatically compose the synthesizable RTL designs of arbitrary cores within a canonical superscalar template. Najaf-abadi et al. [2009] proposed core selectability to incorporate differently designed cores that can be toggled into active employment. They also investigated the overhead and feasibility of multiplexing access to the L1 cache from two different cores.

7. CONCLUSION

Thread-Level Speculation (TLS), by optimistically ignoring infrequent memory aliasing, could improve performance and save energy if the success rate is high. On the other hand, failed speculation could lead to prolonged execution time and increased power consumption, degrading energy efficiency. In this article, we proposed a heterogeneous multicore system to address this issue.

By building an idealistic execution environment and making a number of simplifying assumptions, we were able to efficiently experiment on multicore configurations with a diverse combination of computing and caching components. The resulting design is an architecture that integrates four processing blocks, each containing a four-issue SMT core, a two-issue non-SMT core, and an L1 cache that can be resized between 64KB and 16KB. To cope with various processor reconfiguration overheads, we introduced runtime mechanisms to mitigate their impacts. This was done by reducing the frequency of causal operations. The overhead impact was reduced by 92% as a result. Moreover, we proposed a set of resource allocation schemes that can monitor program execution and match it with the most efficient processor configuration. With all simplifying assumptions removed, the proposed heterogeneous system achieved 18% higher energy efficiency (ED^{2}P) compared to the most efficient homogeneous TLS implementation. It improved performance by 29% and at the same time reduced energy consumption by 3.6% compared to the most efficient homogeneous sequential processor, a 42% improvement in energy efficiency (ED^{2}P).

In the architecture community, speculative threads have long been regarded as energy inefficient. Our work has sent an important message: by combining heterogeneous architecture and speculative thread-level parallelism, we can judiciously allocate on-chip resources and achieve noticeable performance improvement while reducing energy consumption.

REFERENCES

BECCHI, M. AND CROWLEY, P. 2006. Dynamic thread assignment on heterogeneous multiprocessor architectures. In *Proceedings of the 3rd Conference on Computing Frontiers (CF'06)*. ACM, New York, NY, 29–40.

BHATTACHARJEE, A. AND MARTONOSI, M. 2009. Thread criticality predictors for dynamic performance, power, and resource management in chip multiprocessors. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 290–301.

Brooks, D., Tiwari, V., and Martonosi, M. 2000. Wattch: A framework for architectural-level power analysis and optimizations. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY, 83–94.

Chakraborty, K., Wells, P. M., and Sohi, G. S. 2006. Computation spreading: Employing hardware migration to specialize CMP cores on-the-fly. In *Proceedings of the 12th International Conference on Architectural Support for Programming Languages and Operating Systems (APLOS'06)*. ACM, New York, NY, 283–292.

Choudhary, N., Wadhavkar, S., Shah, T., Mayukh, H., Gandhi, J., Dwiel, B., Navada, S., Najaf-abadi, H., and Rotenberg, E. 2012. Fabscalar: Automating superscalar core design. *Micro IEEE 32,* 3, 48–59.

Choudhary, N. K., Wadhavkar, S. V., Shah, T. A., Mayukh, H., Gandhi, J., Dwiel, B. H., Navada, S., Najaf-abadi, H. H., and Rotenberg, E. 2011. FabScalar: Composing synthesizable RTL designs of arbitrary cores within a canonical superscalar template. In *Proceedings of the 38th Annual International Symposium on Computer Architecture (ISCA'11)*. ACM, New York, NY, 11–22.

Eyerman, S. and Eeckhout, L. 2009. Per-thread cycle accounting in SMT processors. In *Proceedings of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, New York, NY, 133–144.

Isci, C. and Martonosi, M. 2003. Runtime power monitoring in high-end processors: Methodology and empirical data. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*. IEEE Computer Society, Washington, DC, 93.

Johnson, T. A., Eigenmann, R., and Vijaykumar, T. N. 2004. Min-cut program decomposition for thread-level speculation. In *Proceedings of the ACM SIGPLAN 2004 Conference on Programming Language Design and Implementation (PLDI'04)*. ACM, New York, NY, 59–70.

Johnson, T. A., Eigenmann, R., and Vijaykumar, T. N. 2007. Speculative thread decomposition through empirical optimization. In *Proceedings of the 12th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'07)*. ACM, New York, NY, 205–214.

Kumar, R., Farkas, K. I., Jouppi, N. P., Ranganathan, P., and Tullsen, D. M. 2003. Single-ISA heterogeneous multi-core architectures: The potential for processor power reduction. In *Proceedings of the 36th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'03)*. IEEE Computer Society, Washington, DC, 81.

Kumar, R., Tullsen, D. M., and Jouppi, N. P. 2006. Core architecture optimization for heterogeneous chip multiprocessors. In *Proceedings of the 15th International Conference on Parallel Architectures and Compilation Techniques (PACT'06)*. ACM, New York, NY, 23–32.

Kumar, R., Tullsen, D. M., Ranganathan, P., Jouppi, N. P., and Farkas, K. I. 2004. Single-ISA heterogeneous multi-core architectures for multithreaded workload performance. In *Proceedings of the 31st Annual International Symposium on Computer Architecture (ISCA'04)*. IEEE Computer Society, Washington, DC, 64.

Liu, W., Tuck, J., Ceze, L., Ahn, W., Strauss, K., Renau, J., and Torrellas, J. 2006. POSH: A TLS compiler that exploits program structure. In *Proceedings of the 11th ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming (PPoPP'06)*. ACM, New York, NY, 158–167.

Lo, J. L., Emer, J. S., Levy, H. M., Stamm, R. L., Tullsen, D. M., and Eggers, S. J. 1997. Converting thread-level parallelism to instruction-level parallelism via simultaneous multithreading. *ACM Trans. Comput. Syst. 15*, 322–354.

Luk, C.-K., Cohn, R., Muth, R., Patil, H., Klauser, A., Lowney, G., Wallace, S., Reddi, V. J., and Hazelwood, K. 2005. Pin: Building customized program analysis tools with dynamic instrumentation. In *Proceedings of the 2005 ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'05)*. ACM, New York, NY, 190–200.

Luo, Y., Packirisamy, V., Hsu, W.-C., and Zhai, A. 2010. Energy efficient speculative threads: Dynamic thread allocation in same-isa heterogeneous multicore systems. In *Proceedings of the 19th International Conference on Parallel Architectures and Compilation Techniques (PACT'10)*. ACM, New York, NY, 453–464.

Luo, Y., Packirisamy, V., Hsu, W.-C., Zhai, A., Mungre, N., and Tarkas, A. 2009. Dynamic performance tuning for speculative threads. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA'09)*. ACM, New York, NY, 462–473.

Luo, Y. and Zhai, A. 2012. Dynamically dispatching speculative threads to improve sequential execution. *ACM Trans. Archit. Code Optim. 9,* 3, 13:1–13:31.

Najaf-abadi, H. H., Choudhary, N. K., and Rotenberg, E. 2009. Core-selectability in chip multiprocessors. In *Proceedings of the 2009 18th International Conference on Parallel Architectures and Compilation Techniques*. IEEE Computer Society, Washington, DC, 113–122.

Open64 Developers. 2001. Open64 compiler and tools. http://www.open64.net.

Packirisamy, V. 2007. Exploring efficient architecture design for thread-level speculation power and performance perspectives. Ph.D. thesis, University of Minnesota - Twin Cities.

Packirisamy, V., Luo, Y., Hung, W.-L., Zhai, A., Yew, P.-C., and Ngai, T.-F. 2008. Efficiency of thread-level speculation in SMT and CMP architectures—performance, power and thermal perspective. In *Proceedings of the International Conference on Computer Design*.

Packirisamy, V., Wang, S., Zhai, A., Hsu, W.-C., and Yew, P.-C. 2006. Supporting speculative multithreading on simultaneous multithreaded processors. In *Proceedings of the 13th International Conference on High Performance Computing*.

Patsilaras, G., Choudhary, N. K., and Tuck, J. 2012. Efficiently exploiting memory level parallelism on asymmetric coupled cores in the dark silicon era. *ACM Trans. Archit. Code Optim. 8,* 4, 28:1–28:21.

Perelman, E., Polito, M., Bouguet, J.-Y., Sampson, J., Calder, B., and Dulong, C. 2006. Detecting phases in parallel applications on shared memory architectures. In *Proceedings of the 20th International Conference on Parallel and Distributed Processing (IPDPS'06)*. IEEE Computer Society, Washington, DC, 88.

Renau, J., Strauss, K., Ceze, L., Liu, W., Sarangi, S., Tuck, J., and Torrellas, J. 2005. Thread-level speculation on a CMP can be energy efficient. In *Proceedings of the 19th Annual International Conference on Supercomputing (ICS'05)*. ACM, New York, NY, 219–228.

Sarangi, S. R., Torrellas, Wei Liu, J., and Zhou, Y. 2005. ReSlice: Selective re-execution of long-retired misspeculated instructions using forward slicing. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE Computer Society, Washington, DC, 257–270.

Sazeides, Y., Kumar, R., Tullsen, D. M., and Constantinou, T. 2005. The danger of interval-based power efficiency metrics: When worst is best. *IEEE Comput. Archit. Lett. 4*, 1.

Shivakumar, P. and Jouppi, N. 2001. CACTI 3.0: An integrated cache timing, power and area model. Tech. rep., Compaq Computer Corporation.

SimpleScalar LLC. 2004. The SimpleScalar tool set. http://www.simplescalar.com/.

Sohi, G. S., Breach, S. E., and Vijaykumar, T. N. 1995. Multiscalar processors. In *Proceedings of the 22nd Annual International Symposium on Computer Architecture (ISCA'95)*. ACM, New York, NY, 414–425.

Steffan, J. G., Colohan, C., Zhai, A., and Mowry, T. C. 2005. The STAMPede approach to thread-level speculation. *ACM Trans. Comput. Syst. 23*, 253–300.

Steffan, J. G., Colohan, C. B., Zhai, A., and Mowry, T. C. 2000. A scalable approach to thread-level speculation. In *Proceedings of the 27th Annual International Symposium on Computer Architecture (ISCA'00)*. ACM, New York, NY, 1–12.

Suleman, M. A., Mutlu, O., Qureshi, M. K., and Patt, Y. N. 2009. Accelerating critical section execution with asymmetric multi-core architectures. In *Proceeding of the 14th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'09)*. ACM, New York, NY, 253–264.

Tuck, J., Liu, W., and Torrellas, J. 2007. Cap: Criticality analysis for power-efficient speculative multithreading. In *Computer Design, 2007. ICCD 2007. 25th International Conference on*. 409–416.

Tullsen, D. M., Eggers, S. J., Emer, J. S., Levy, H. M., Lo, J. L., and Stamm, R. L. 1996. Exploiting choice: Instruction fetch and issue on an implementable simultaneous multithreading processor. In *Proceedings of the 23rd Annual International Symposium on Computer Architecture (ISCA'96)*. ACM, New York, NY, 191–202.

Vijaykumar, T. N. and Sohi, G. S. 1998. Task selection for a multiscalar processor. In *Proceedings of the 31st Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'98)*. IEEE, Los Alamitos, CA, 81–92.

Wang, H.-S., Zhu, X., Peh, L.-S., and Malik, S. 2002. Orion: A power-performance simulator for interconnection networks. In *Proceedings of the 35th Annual ACM/IEEE International Symposium on Microarchitecture (MICRO'02)*. IEEE, Los Alamitos, CA, 294–305.

Wang, S. 2007. Compiler techniques for thread-level speculation. Ph.D. thesis, University of Minnesota, Twin Cities.

Wang, S., Dai, X., Yellajyosula, K. S., Zhai, A., and chung Yew, P. 2005. Loop selection for thread-level speculation. In *Proceedings of the 18th International Workshop on Languages and Compilers for Parallel Computing*.

Wu, Q., Juang, P., Martonosi, M., and Clark, D. W. 2005a. Voltage and frequency control with adaptive reaction time in multiple-clock-domain processors. In *Proceedings of the 11th International Symposium on High-Performance Computer Architecture*. IEEE Computer Society, Washington, DC, 178–189.

WU, Q., MARTONOSI, M., CLARK, D. W., REDDI, V. J., CONNORS, D., WU, Y., LEE, J., AND BROOKS, D. 2005b. A dynamic compilation framework for controlling microprocessor energy and performance. In *Proceedings of the 38th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO'05)*. IEEE Computer Society, Washington, DC, 271–282.

YANG, S.-H., FALSAFI, B., POWELL, M. D., AND VIJAYKUMAR, T. N. 2002. Exploiting choice in resizable cache design to optimize deep-submicron processor energy-delay. In *Proceedings of the 8th International Symposium on High-Performance Computer Architecture (HPCA'02)*. IEEE Computer Society, Washington, DC, 151.

ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. 2002. Compiler optimization of scalar value communication between speculative threads. In *Proceedings of the 10th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'02)*. ACM, New York, NY, 171–183.

ZHAI, A., COLOHAN, C. B., STEFFAN, J. G., AND MOWRY, T. C. 2004. Compiler optimization of memory-resident value communication between speculative threads. In *Proceedings of the International Symposium on Code Generation and Optimization (CGO'04)*. IEEE Computer Society, Washington, DC, 39.

Received June 2012; revised February 2013; accepted August 2013