CHILL: a system for fine-grained mapping of chained high impact long-latency load phases on tightly coupled heterogeneous multi-cores

Robert Chen* and Glenn Reinman

Department of Computer Science,
University of California,
Los Angeles, CA, USA
Email: rchen@cs.ucla.edu
Email: reinman@cs.ucla.edu
*Corresponding author

Abstract: With increasing power and application demands, heterogeneous multi-core processors are becoming more prevalent. However, the key to proper utilisation of heterogeneous multi-cores is assigning, or mapping, the right application to the right core type. Recent work has shown that fine-grained mapping takes advantage of short program phases with highly variant performance requirements, and can elicit greater benefits from tightly coupled heterogeneous multi-cores. In this paper, we show that bottlenecks in performance can occur in fine-grained program phases during chains of high impact long-latency loads. We design a system that detects these bottleneck phases, and propose accelerating these phases on the out-of-order core for better performance and energy efficiency. Our system operates within 10% of performance, and 2.6% of energy to an oracle resource mapper. This translates to a 44.4% performance gain, and 9.2% energy savings over existing fine-grained mapping techniques.

Keywords: heterogeneous multi-cores; tightly coupled multi-cores; resource mapping; fine-grained mapping; fine-grained scheduling; bottleneck phases.


Biographical notes: Robert Chen is a PhD candidate in the Department of Computer Science at the University of California, Los Angeles. He received his MS in Computer Science and Engineering at the University of California, San Diego, and BS in Electrical Engineering at the University of California, Los Angeles.

Glenn Reinman is a Professor in the Department of Computer Science at the University of California, Los Angeles. He received his PhD and MS in Computer Science at the University of California, San Diego in 2001. He received his BS in Computer Science and Engineering from the Massachusetts Institute of Technology in 1996.

1 Introduction

Heterogeneous multi-core processors have been shown to be more energy efficient and sometimes better in performance than homogeneous multi-cores (Bower et al., 2008; Kumar et al., 2003, 2004). With mobile devices becoming increasingly mainstream, the energy efficient characteristics of heterogeneous multi-cores make them especially compelling for device manufacturers. Heterogeneous designs are realities today in the form of ARM Ltd.’s big.LITTLE cores (ARM, 2013), and NVIDIA’s (2015) Tegra X1. The big.LITTLE design features performance oriented but power hungry ‘big’ out-of-order (OoO) cores, combined with ‘LITTLE’ power efficient but reduced performance in-order cores. The Tegra X1 also features eight big.LITTLE cores along with 256 power efficient GPU cores.

A key factor in obtaining energy and performance efficiency on heterogeneous multi-cores is the use of scheduling program phases on cores which are suited to phase characteristics. Prior works have experimented with scheduling in a coarse-grained manner with epochs of millions of instructions (Becchi and Crowley, 2006; Shelepov et al., 2009; Koufaty et al., 2010; Craeynest et al., 2012; Chen and John, 2009). However, recent work by Lukefarh et al. (2012) and Padmanabha et al. (2013) show that shorter phases of thousands or hundreds of instructions can achieve greater gains in performance and energy savings than coarse-grained scheduling. This is because
fine-grained phases can experience greater performance variance which allows more opportunities for optimisation. Coarse-grained policies are less variant in performance because longer phases experience more instruction level parallelism (ILP) that amortise and hide the costs of expensive instructions. For our work, we will refer to ‘scheduling’ as ‘resource mapping’, in order to differentiate ourselves from the OS schedulers of prior work.

An important type of fine-grained program behaviour occurs during phases where chains of high impact long-latency loads (CHILLs) become bottlenecks for performance. CHILLs are last-level L2 misses which are data dependent on each other, and accumulate dependent instructions (‘shadows’) which wait in the instruction window for data to return from memory. When the data become available, the backlog of dependent instructions can execute with high ILP, which can be exploited for performance and energy gains. Consequently, CHILL phases require more memory resources for tolerating the loads themselves, and more ILP to amortise the costs of waiting dependent instructions.

Our system finds bottleneck CHILL phases in hardware at runtime, and accelerates them on the OoO core of a heterogeneous processor. We track chains of dependencies between long-latency loads (LLLs) and instructions dependent on them, in order to find CHILLs. The chain information is stored and when the chain is encountered again in the future, the program migrates to the OoO core. While the program is executing during the CHILL phase on the OoO core, our system will migrate the program between the OoO and in-order cores, depending on the amount of shadow instructions it encounters. Our contributions are as follows:

- We demonstrate that phases of CHILLs present fine-grained bottlenecks in programs, and have performance behaviour which can be exploited.
- We design a system in hardware to find and track CHILL phases at runtime.
- We use our system to predict and accelerate programs through CHILL phases on a heterogeneous multi-core by properly mapping resources.

The remainder of the paper is organised as follows: Section 2 provides a detailed motivation for our work, Section 3 discusses related work, Sections 4 and 5 discuss the design of our system and methodology, Section 6 explains the results, and Section 7 concludes our work.

2 Motivation

2.1 Fine-grained resource mapping

Fine-grained resource mapping can be advantageous on tightly-coupled architectures because it can utilise performance and energy differences of a general-purpose heterogeneous core more efficiently. Shorter phases (Figure 1) experience more varied and extreme program IPC than longer phases (Figure 2), which create more opportunities for performance and energy optimisation. Figures 1 and 2 show the IPC of the mcf benchmark for 100 K instructions, in periods of 500 and 10 K instructions, respectively. The IPC of the short phases in Figure 1 vary from 0.15–1.7, whereas the longer phases in Figure 2 vary from 0.15–0.4. Longer phases exhibit more stable IPC because costly instructions are amortised, and consequently are unable to realise the benefits of more fine-grained optimisation inherent in programs. Even for moderately sized phases of 1,028 instructions or more (Figure 15), the performance levels off while energy savings decrease, which means anticipating and mapping short phases properly is important. Short phases also incur more switching, which obviates the need for tightly-coupled cores with low switching overhead (Section 5).

Figure 1  Mcf, variant IPC, 500 instr epochs (see online version for colours)
2.2 CHILL phases

LLLs, or loads which miss in the last level of cache (L2 data misses), have often been found to be sources of bottlenecks in programs (Shelepov et al., 2009; Collins et al., 2001; Abraham and Rau, 1994). Chains of LLLs which depend upon each other are also important bottlenecks in program performance because they consume memory bandwidth, MSHR entries, and other instructions dependent on them consume space in the instruction window. These chains of LLLs also attract more dependent non-load ‘shadow’ instructions than normal individual LLLs (Figure 4), making these LLLs ‘high impact’. When the CHILLs have their data fulfilled from memory, the pipeline is clogged with an abundance of shadow instructions waiting to be executed. Our system identifies these phases, and accelerates them on the OoO core in order to ‘burst’ the program through these bottlenecks.

To differentiate CHILLs from other individual LLLs, we quantified CHILL characteristics across the SPEC2006 benchmarks. CHILLs have geometric shadows of 3.15 instructions, compared to individual LLL shadows of 2.1. We find that the IPC of CHILL phases is 9.58× worse (geometric) than that of non-CHILL phases (Figure 3), and that programs can spend a geometric mean of 21% of cycle time in CHILL phases. The combination of reduced IPC and the amount of time spent on CHILL phases means that efficiently executing through these phases is worthwhile.

CHILL phases refer to both epochs in which either actual CHILLs or their dependent shadow instructions occur, while non-CHILL phases have neither. This is illustrated in Figure 5, where a period of 20 epochs is shown from mcf. The first three epochs contain CHILLs, and
experience the lowest IPC. The next 14 epochs do not contain CHILLs, but experience variant IPC because they contain shadow instructions dependent on the CHILLs. Knowing this, it is important to accelerate regions beyond the epochs in which CHILLs themselves occur, in order to capture the entire CHILL phase.

During a CHILL phase, we distinguish between LLLs and their shadows, and seek to accelerate the shadows on the OoO, while reactively mapping to the in-order core when shadows are infrequent. This potentially allows the actual LLLs in CHILL phases to be run on the in-order for energy savings, although most CHILL LLLs occur in the same epoch as their shadows. Other types of LLLs, e.g., unchained individual L2 misses, are run on the in-order core for energy savings, and are identified during the ridge regression analysis in our base reactive mechanism in Section 4.6.

To determine which type of core is better suited to running CHILL phases, we tested their performance on OoO and in-order cores (Figure 6). CHILL phases perform better on the OoO core, with a 33.8% geomean increase in IPC over the in-order core. Although some benchmarks like astar, bzip, and gcc exhibit small losses in IPC on the OoO, we are still able to achieve reasonable gains, as will be discussed in Section 6.

**Figure 4** Shadow instr differences (see online version for colours)

**Figure 5** Snapshot of mcf, certain non-CLL phases require acceleration (see online version for colours)
3 Related work

3.1 Load criticality and dependence

Part of our work relies on finding LLLs at runtime, and we highlight the work that is more closely related to our analytical techniques. Collins et al. (2001) speculatively pre-compute slices of programs which lead to delinquent loads, which results in reduced wait times for future data cache misses. Panait et al. (2004) refine the static technique for identifying delinquent loads via basic block profiling in post-compilation. Both techniques rely on identifying possible ‘delinquent’ loads as critical during compilation or post-compilation. Our system identifies dynamic LLLs and correlates them directly with criticality at runtime.

Srinivasan et al. (2001) dynamically identify critical loads in hardware based on various factors. They classify critical loads into three categories:

1. loads which feed mispredicted branches
2. loads which feed other loads that miss in L1
3. loads in which the number of independent instructions within a window of instructions is below a certain threshold.

With this criticality information, they modify the cache to speed up critical loads in a manner similar to that of a victim cache. Our analysis differs in that we find critical phases during chains of high impact L2 load misses to be bottlenecks, and take advantage of modern heterogeneity of processors to accelerate through bottleneck phases. Furthermore, our solution for utilising this information is different in that we take advantage of modern heterogeneity in processors to accelerate through critical phases while saving energy.

Another class of work finds data dependencies at runtime in hardware, but their techniques differ from our system in various aspects. Roth et al. (1998) create a prefetching scheme based on finding chains of pointer-based loads for linked data structures. Their system includes tables for loads in progress and known dependencies. However, their technique is limited to identifying dependencies between two instructions, rather than long chains of loads. Raasch et al. (2002) enhance the performance of the instruction queue with dependency lanes. Each lane represents a chain of instructions dependent on a load to a particular register. Instructions are sniffed for source dependencies, and added to chains with estimated delays until issue time. Our mechanism differs in that we keep simpler bit-vectors to store dependence chains, and perform merging of chains to save space. Chen et al. (2003) create a system for tracking data dependencies dynamically for enhancing branch prediction. They use a FIFO of bit-vectors to track dependencies of all running instructions. Our system differs in that we only track LLLs and their dependent instructions, and remove entries when the lifetime of the LLLs end.

3.2 Heterogeneity and scheduling

Prior work shows the benefits of heterogeneous cores for both energy efficiency and performance (Bower et al., 2008; Kumar et al., 2003; Kumar et al., 2004; Lukefahr et al., 2014). An abundance of other work use various metrics to schedule programs statically (Chen and John, 2009; Shelepov et al., 2009) or dynamically (Becchi and Crowley, 2006; Craeynest et al., 2012; Koulaty et al., 2010; Costa et al., 2009). These techniques operate at granularities from millions of instructions to milliseconds of time, and often operate in the OS (Muthukaruppan et al., 2014) or even at the user-level (Petrucci et al., 2015; Mor and Maillard, 2011).

For finer granularities, Lukefahr et al. (2012, 2016) and Padmanabha et al. (2013) show that thousands or hundreds of instructions can achieve more gains, which is the
computing substrate on which we focus. Lukefahr proposed a tightly coupled multi-core with two backends, one OoO and one in-order, which can be dynamically switched at runtime. Their mapping algorithm is based on an a priori ridge regression analysis of core metrics, and uses the runtime core metrics to make decisions based on the prior analysis. With this, they achieve fine-grained mapping at 300 (Lukefahr et al., 2012) and 1,000 (Lukefahr et al., 2016) instructions per mapping interval. Padmanabha’s work extends Lukefahr’s by finding program phases with branch history signatures. They coalesce pieces of past branch target addresses to mark loop starting points, and associate these points with runtime performance. Our tightly-coupled heterogeneous core model is similar to that in Lukefahr et al. (2012), while our mapping granularity is similar to the trace-based work (Padmanabha et al., 2013) at hundreds of instructions. However, we differ by extending the mapping algorithm with CHILL analysis, and we do not use the branch history signatures in the trace-based system. In DynaMOS (Padmanabha et al., 2015), the same authors create a technique for memoising OoO schedules that are consistently repetitive, and running them on the in-order core for energy efficiency. By capturing and storing certain OoO instruction sequences, they can replay these sequences on the in-order core for better energy efficiency. Our system differs from DynaMOS in that it discerns problematic phases, accelerates these phases accordingly, and performs reactive mapping in other phases.

Another proposed fine-grained mapping scheme is a morphing core (Srinivasan et al., 2013). Srinivasan et al. (2013) create an architecture similar to Lukefahr’s, however, it contains one backend that changes between OoO and in-order configurations on-the-fly. A transition from OoO to in-order mode requires turning off parts of the fetch and decoding logic, LSQ, ROB, RAT, and some execution units. They base their switching mechanism on a power/Watt metric, and switch in intervals of 500 instructions. Our scheme differs in that we use two backends, one for each OoO and in-order cores, with shared fetch logic. We also differ in that our switching mechanism is based on both CHILL analysis and performance metrics, on which we perform a priori ridge regression analysis, and adapt at runtime.

4 CHILL system and algorithms

Overall, our hardware system captures retiring instructions, tracks their dependencies in a table of bit-vectors which correspond to active registers, and finds and records CHILL phases from the tables of dependencies. As instructions retire, their source and destination registers are mapped in bit-vector tables, whose entries correspond to pending LLLs. This creates a simple dependency analysis methodology in hardware, which is similar to the one used in Chen et al. (2003), albeit modified to store bit-vectors for LLLs (Section 4.2). At the end of every 512-instruction epoch, the table is analysed to determine which LLLs are parts of CHILL phases (Section 4.3). After a CHILL phase ends (the LLL registers are overwritten with other data), our system ‘remembers’ the phase by storing its starting point (last backward branch PC), and keeps a count of the duration of epochs (Section 4.4). This stored CHILL information is then used to predict when these phases recur in the future (Section 4.5), and are mapped accordingly to the OoO core. When a stored last backward branch PC is encountered, the system estimates that a CHILL phase will begin soon, and maps the program to the OoO core. During the CHILL phases, we track shadow instructions and migrate between the OoO and in-order cores in a semi-reactive manner: encountering more shadows keeps the program on the OoO core, while fewer shadows initiate migration to the in-order core. In non-CHILL phases, we default to a standard reactive mapping mechanism, which can map non-CHILL LLL phases to the in-order core, among other optimisations. The following subsections describe the hardware components and algorithms in detail.

4.1 System components

The tightly coupled multi-core architecture (Figure 7) used contains one frontend fetch engine, with two execution backends (OoO and in-order). Both backends feed their runtime statistics, described in Section 4.6, into the CHILL system, which then informs the fetch logic of core mapping decisions. The CHILL system uses three tables (Figure 8) to detect and manage chains of LLLs: current dependencies table [CDT, Figure 8(a)], pending chains table [PCT, Figure 8(b)], and complete chains table [CCT, Figure 8(c)]. As instructions retire, they are analysed in the manner described in Section 4.2 for dependencies to LLLs in the CDT. When a kill of a LLL occurs, certain entries are removed from the CDT, and placed into the PCT. The PCT holds chains of LLLs that are being constructed and have not all been killed. When a chain in the PCT ends, its information is passed into the CCT, which aids in predicting CHILL phases. In essence, the CDT tracks all LLL dependencies, the PCT tracks ongoing CHILL phases (graduated from the CDT), and the CCT remembers the completed CHILL phases (graduated from the PCT) for future prediction.

4.2 Dependence tracking (populating the CDT)

The CHILL dependence analysis relies on observing relations between instructions’ source and destination registers. Figure 9 provides an illustrative sample of source code, with bolded instructions being LLLs. We track the dependencies in a table of bit-vectors, seen in Figures 9(a), 9(b), 9(c), and 9(d). Each of Figures 9(a), 9(b), 9(c), and 9(d) represent the entire CDT at various points in the program, with row entries numbered in increasing order from bottom to top. The 64 CDT entries contain a 64-bit dependence bit-vector, a 10-bit last backward branch PC, and a 5-bit start of epoch entry.

To begin, Figure 9(a) represents the CDT after all of the instructions in the sample code have been retired in order. Each row in the table of bit-vectors [Figure 9(a)] represents
the dependencies for the $N^{th}$ register; the first row (bottom row) represents the dependencies for register 1, etc. When a LLL is encountered, like instructions A, D, and E, we set the $N^{th}$ bit in the $N^{th}$ row to 1 in order to signify a LLL dependence (where $N$ is the destination register number). For example, instruction A in Figure 9 sets the first bit of the first bit-vector to 1. Instruction D sets the 4th bit in the 4th bit-vector to 1, and because the result is dependent on register 1, it stores the union of bit-vectors 1 and 4 (1001) in row 4.

Other shadow instructions which depend on the LLLs will add their dependencies to the bit-vector table, such as instructions B, C, F, G, and H in Figure 9. To create the dependence bit-vector for every instruction, we use the instruction’s source register numbers and index into those rows in the CDT. We then compute the union of those source rows in order to find all of the other registers the instruction is dependent upon. Then, we use the instruction’s destination register number to index into the CDT, and compute the union of the previous source register rows with the destination row.

The entire union of bit-vectors is stored in the row represented by the destination register. For example, in Figure 9(a), instruction H creates an entry of 11001 in the 8th row of the table. The union of the 4th and 5th rows is computed, because the instruction’s source registers are r5 and r4. This union is then stored in the 8th entry, which signifies that r8 depends on LLLs to r1, r4, and r5. Figure 10 shows the CDT filled with the example code in Figure 9. If the CDT is modified with a shadow instruction, the system indicates to the prediction mechanism in Section 4.5 that the epoch has encountered high impact shadows.

---

**Figure 7** Two backends, one fetch engine, with backends communicating with CHILL system (see online version for colours)

**Figure 8** CHILL system: 64 64-bit bit-vectors (CDT), five PCT, and five CCT entries, (a) CDT (b) PCT (c) completed chains table (CCT) (see online version for colours)

**Figure 9** Example of code analysis during CHILL phase

<table>
<thead>
<tr>
<th>Program Order</th>
<th>A. ld r1 imm</th>
<th>B. add r2 r1 r3</th>
<th>C. add r3 r2 r4</th>
<th>D. ld r4 r1</th>
<th>E. ld r5 r4</th>
<th>F. add r6 r4 r7</th>
<th>G. add r7 r6 r8</th>
<th>H. add r8 r5 r4</th>
</tr>
</thead>
<tbody>
<tr>
<td>entry #</td>
<td>8. 11001</td>
<td>7. 1001</td>
<td>6. 1001</td>
<td>5. 11001</td>
<td>4. 1001</td>
<td>3. 1</td>
<td>2. 1</td>
<td>1. 1</td>
</tr>
<tr>
<td></td>
<td>10000</td>
<td>0</td>
<td>1001</td>
<td>10000</td>
<td>0</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>0</td>
<td>1001</td>
<td>1</td>
<td>11000</td>
<td>1000</td>
<td>1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Chains of LLLs can be identified as $N$th rows whose $N$th bit is set to 1, and which have other bits within the bit-vector set to 1, such as rows 4 and 5 in Figure 9(a). Rows 2, 3, 6, 7, and 8 do not represent CHILLs because their $N$th bits are not set to 1. Although row 1 has only its first bit set to 1 and may seem like an independent LLL, rows 4 and 5 also have their first bits set to 1, and hence r1 is part of the CHILL of 1, 4, and 5. It should be noted that only the occurrence of LLLs can begin the population of the bit-vectors. The core can identify loads as being long-latency if they do not return a result within the L2 hit time, which is 20 cycles in our system.

Bit-vector entries are cleared when a LLL is killed. Figure 9(b) shows the result of register 4 being killed; only subsets of bit-vectors matching exactly the 4th bit-vector are set to 0. Figure 9(c) shows a superset bit-vector kill, in which only exactly matching supersets of the 5th bit-vector are set to 0 in the table. Finally, Figure 9(d) shows the result of a single LLL dependency being killed in register 1, where all subsets are set to 0.

4.3 Collecting and merging CHILLs (CDT to PCT)

When a LLL’s register is killed in the CDT, we begin to form chains in the PCT [Figure 8(b)]. The PCT contains five entries of 64-bit bit-vectors which hold chain register identities, and the corresponding 10-bit last backward branch PCs, and a 5-bit start epoch of the first LLL in the chain. The current full chain entry is a temporary store for the remaining unkill LLL registers that the entire chain is waiting on.

Referring back to Figure 9 illustrates what occurs in the CDT during kills of LLLs, and how they are graduated to the PCT. Figure 9(a) shows the state of the CDT after retiring all instructions in the example code before any kills occur. Upon a LLL kill, we first examine the CDT entry being killed, and traverse the CDT to search for other entries containing the killed register’s identity, and place this result into the current full chain entry of the PCT. In doing so, we attempt to find the longest current chain for which the killed entry is a member. For example, in step 1 of Figure 11, if r4 is killed, the system uses row 4 and begins searching from row 1. Row 1 does not contain a 1 in the fourth bit position (because r4 is being killed), so row 1 is left untouched. The system continues until it reaches row 5, which has a 1 in the 4th bit position, meaning that r5 somehow depends on r4. Here, row 4 is unioned with row 5, and the result is saved in the current full chain entry for further iterations in the CDT. Traversing rows 6, 7, and 8 result in three more unions because all three rows contain a 1 in the 4th bit position. After traversing the entire CDT, the final result of all the unions is 11001, and becomes the current full chain entry. During traversal, the entire sub-bit-vector of r4 (1001) is removed from any row which contains a perfect match, resulting in a CDT that looks like Figure 9(b). In other words, only the 1001 pattern is removed from every CDT row because that was the entry in
row 4 at the time of r4’s killing. Notice that killing any of the LLLs in Figure 9 will result in a current full chain entry of 11001.

Next, the system attempts to add the current full chain entry into the PCT, seen in step 2 of Figure 11. If the PCT already contains entries, we search the PCT for any matches with any of the valid bits in the current full chain. If a match occurs, signifying a pending chain merge, the matching PCT bit-vector entry is updated with the union of its bit-vector and the current full chain. If no current PCT entries match any of the valid bits in the current full chain, a new PCT entry is created with the current full chain as the bit-vector. The corresponding backward branch PC, and start epoch from the killed CDT entry are copied into the new PCT entry. Finally, we kill the actual identity bit in the corresponding PCT bit-vector. For example, killing r4 will result in a PCT bit-vector of 10001, which means that r4’s load dependencies have ended, and that registers 5 and 1 remain active in the chain.

4.4 CHILL tracking (PCT to CCT)

The CCT holds CHILL phases which have been completed, which are graduated pending chains from the PCT. Each of the five CCT entries is represented by a 10-bit last backward branch PC, a 5-bit duration entry, and a 5-bit countdown entry. After killing a PCT entry, we check whether that entry became 0, which signifies that a pending chain has ended with no more LLLs, and is ready to become a completed chain in the CCT. If so, a chain has ended and we must update the CCT in Figure 8(c). We first check whether the killed PCT’s backward branch PC matches any entries in the CCT. If there is a match, this indicates that an already completed chain is being updated, and we update the CCT entry with the oldest backward branch PC, and the longer duration. The duration is calculated by subtracting the start epoch from the current epoch number during the kill. If there is no matching CCT entry, we create a new one by moving the killed PCT’s last backward branch PC, and duration into the CCT. Finally, the countdown entry in the CCT is set to 0.

4.5 Prediction mechanism

The prediction mechanism utilises the information in the CCT, and whether the program is encountering high impact shadow instructions. The mechanism runs as instructions are retired, and predictions will vary depending on which core the program is currently located. Overall, the system tries to execute the program on the OoO when it encounters shadows during CHILL phases, and uses a standard reactive prediction mechanism during non-CHILL phases.

When a program begins, there will be no completed chains, and the system will have no knowledge of pending chains. To mitigate such ‘cold start’ phases, we check every epoch whether the CDT has created a PCT entry, or if a CDT entry was modified with non-load shadow dependencies. If this occurs, we migrate the program to the OoO core for the next epoch. This technique allows us to run more optimally during cold start by running on the OoO when shadow instructions dependent on CHILLs have an impact.

As a program executes, it will populate the PCT and CCT with pending chains in progress and completed chains, respectively. Every CCT entry contains a duration field which contains the number of epochs from the first LLL in the CHILL phase to when the last LLL in the chain is killed. We detect whether the program is within a completed chain by comparing branch target PCs of retiring instructions to the last backward branch PCs recorded in the CCT. If there is a match, then the program has entered a CHILL phase, and the program is transitioned to the OoO core for the next epoch. Upon entering a completed chain, the system sets the countdown field of the CCT entry to match the duration field. Every succeeding epoch, the countdown is decremented by 1. Multiple completed chains can be active at once, and their countdowns are decremented simultaneously.

While a program is running when there are positive countdowns in the CCT or active pending chains in the PCT, the program will be active on the OoO core during CHILL phases, but we also need to carefully switch to the in-order core during opportune moments for better energy efficiency. To take advantage of in-order energy savings during active pending chains and live completed chains, we use a 4-bit saturation counter that counts the number of high impact shadow instructions. If shadow instructions are encountered during the epoch, the saturation counter is incremented by 1, and decremented otherwise. While executing on the OoO, when the saturation counter falls below half of its maximum value, we estimate that the program has not modified an active chain for a significant period of time, and switch execution to the in-order core. If a CHILL is encountered while on the in-order core, the saturation counter is set to its maximum value and the program switches to the OoO in anticipation of new shadows. In this way, we attempt to maximise performance and energy savings during CHILL phases while in the presence of active chains.

4.6 Base reactive mechanism

During non-CHILL phases, we revert to a base reactive mechanism which depends on measurements of MLP, ILP, L2 miss and hit rates, and branch misprediction rates, similar to the one used in Lukefahr et al. (2012). For ILP and MLP measurements, we use a system similar to one described in Chen et al. (2003), and leverage information in our own CDT. Prior to experimental runs, we gathered data on the first 10 millions instructions of our benchmarks with the SPEC 2006 test input. We perform a ridge regression analysis on the metrics to determine a correlation between these statistics on the OoO core and the in-order core’s performance, and vice versa. Figure 12 shows the results of our ridge regression analysis, and how OoO and in-order statistics correlate to the performance of their counterparts.
With this profiling information, we estimated the performance of the non-active core at runtime, and map the program reactively using a standard proportional-integral controller mechanism like the one in Lukefahr et al. (2012). Equation (1) shows the proportional-integral used to calculate a mapping decision while currently running on the OoO core. We track the total CPI_{observed} and all past errors in CPI_{pasterror}, as part of the integral term. Our proportional term is based on the estimate of CPI on the in-order core CPI_{in-orderedesimated}, and the error of that estimate, CPI_{currenterror}, which comprises of the difference between CPI_{in-orderedesimated} and the observed CPI of the current epoch.

\[ \Delta CPI_{OoOdecision} = \sum CPI_{observed} + \alpha \sum CPI_{pasterror} + \beta CPI_{currenterror} - CPI_{in-orderedesimated} \]  

(1)

The decision is made depending on the result of \( \Delta CPI_{OoOdecision} \). If the result is negative, that means that the estimated in-order CPI will be better, which indicates that the system should map to the in-order core. If the result is positive, the overall CPI experienced on the OoO core is better than the estimated in-order CPI, indicating that the program should remain on the OoO.

### 5 Methodology

We model our tightly-coupled heterogeneous core similar to the composite core described in Lukefahr et al. (2012), in which the OoO and in-order cores are modelled after ARM’s big.LITTLE (Greenhalgh, 2012). The big OoO core is modelled after the Cortex-A15, which is three-way issue with a pipeline depth of 15–25 stages. The LITTLE in-order core is modelled after the Cortex-A7, which has a shorter pipeline of 8–10 stages with two-way issue capability, depending on instruction dependencies.

In the tightly-coupled design, the big and LITTLE share L1 and L2 caches, a fetch unit, branch predictor, and the CHILL system. Each core has its own decode and OoO/in-order execution engines. Switching, or migrating, from the big to the LITTLE requires draining the OoO pipeline (commit as many instructions as possible, and flush non-commitable instructions), and traversing the RAT to restore values from reservation stations into the architecturally visible registers. The vice versa occurs during LITTLE to big transitions. The switching delay depends on the number of instructions in the pipeline and the amount of register state which needs to be preserved, which requires 30 cycles on average (Lukefahr et al., 2012). Current OS switching on big.LITTLE takes 20 ms (Greenhalgh, 2012). Although it is possible to clock gate the inactive part of the core, we opt for the most pessimistic power consumption model, in which we account for both big and LITTLE static power during the entirety of execution.

Our CHILL system requires 5615 bits in table space for the CDT/PCT/CCT. Using CACTI (Shivakumar and Jouppi, 2001), we find that our design requires 0.023 mm² of area, which is negligible overall. The complete system is accessed on every CHILL kill, and once at the end of every epoch. Non-load instructions access only the CDT. This leads to an estimated power consumption of 0.0433 W.

We run programs from SPEC 2006 (Henning, 2006) in the Simics (Magnusson et al., 2002) and GEMS (Martin et al., 2005) simulators. Core power and energy are modelled in McPAT (Li et al., 2009). Table 1 provides more details on the architectural parameters simulated. All simulations execute over 15 million instructions, with 512-instruction epoch size, and compiled for SPARC v9 with –O3 optimisations.

<table>
<thead>
<tr>
<th>Table 1</th>
<th>Simulation core parameters</th>
</tr>
</thead>
<tbody>
<tr>
<td>OoO core</td>
<td>3-wide issue, 15–25 stage pipeline, 128 entry</td>
</tr>
<tr>
<td>(1 GHz)</td>
<td>ROB, 160 entry reg file</td>
</tr>
<tr>
<td>In-order core</td>
<td>2-wide issue, 8–10 stage pipeline, 64 entry reg file</td>
</tr>
<tr>
<td>(1 GHz)</td>
<td>Memory</td>
</tr>
<tr>
<td>32kB L1 I-cache (4 MSHR), 32 kB L1 D-cache system (4 MSHR), 2 MB L2 cache (8 MSHR), 4 GB Shared RAM</td>
<td></td>
</tr>
</tbody>
</table>

---

*Figure 12* Ridge regression analysis coefficient composition
6 Results and analysis

6.1 Mapping mechanisms

We compare CHILL against other types of fine-grained resource mapping systems. First, we compare against an oracle resource mapper, which tolerates a 5% decrease in performance while seeking opportunities for energy efficient computation. Second, we compare with a reactive mapper with characteristics described in Section 4.6. It dynamically correlates core performance to certain core statistics gathered at runtime. Prior to simulation, the MLP, ILP, L2 hit and miss rates, and branch misprediction characteristics are correlated to performance in pure OoO and in-order runs. The mapping is then included in the system at runtime for reactive mapping. Third, we compare against a reactive mapper with a 4-bit saturation counter to help smooth perturbations inherent to a purely reactive scheme. Finally, we compare with the trace-based performance prediction scheme proposed in Padmanabha et al. (2013). This scheme relies on creating identifiers for different phases of execution by combining pieces of past backward branch PCs, and phases are correlated with performance at runtime. Although there are energy improvements to the trace-based system in DynaMOS (Padmanabha et al., 2015), we find that the small in-order phases of DynaMOS are not captured frequently enough to make a significant impact on the trace-based system, so we elect to compare with trace-based without DynaMOS. Both the saturated and purely reactive systems we test are similar to the reactive system in the trace-based scheme. The advantage of the trace-based scheme is in its identification of phases, while the reactive systems do not contain such a feature.

6.2 Performance

Figure 13 shows IPC normalised to the oracle for all four mapping schemes. Overall, the CHILL system operates at a 10% loss to the oracle, while the trace-based system operates at a 37.8% loss. The saturated counter scheme loses 42.5%, while the reactive scheme loses 42.9%. For the hmmer benchmark, the trace-based system experiences a 1% performance loss compared to the saturated counter and reactive systems. This is because hmmer contains more branch mispredictions which cause more core switches than necessary because the trace-based system uses branch PCs as phase markers. This also explains the libq drop in performance for the trace-based system. Here, the saturated counter and reactive systems experience about a 7.5% increase in performance than trace-based. The overall performance could be improved for CHILL phases with more serious branch misprediction rates, as in libq and mcf. Further details in performance are also explained in Section 6.3

6.3 Energy

Energy savings normalised to the oracle are shown in Figure 14. Overall, the CHILL system operates at a 2.6% loss in energy compared to the oracle. The trace-based system experiences a 10.8% loss, while the saturated counter and reactive schemes experience 21.6%, and 21.7% energy losses compared to the oracle, respectively. There are a few instances of systems beating the oracle on energy savings. The astar benchmark experiences relatively few CHILL phases, which means there are more opportunities for energy-saving in-order execution. Both the CHILL and trace-based systems detect this, with the trace-based system gaining 4% more energy savings than CHILL. The saturated counter and reactive systems also detect the increased opportunities for energy savings, but save about 10% less energy than the CHILL and trace-based systems. The price for saving energy on astar is decreased performance compared to the oracle, as seen in Figure 13.

Figure 13  IPC normalised to oracle (see online version for colours)
The gcc benchmark experiences phases with many CHILLs, but mixed with high branch misprediction rates during those CHILL phases. This causes the CHILL system to run unnecessarily on the OoO during the CHILL phases, while the trace-based, saturated counter, and reactive systems run with more energy efficiency on the in-order. They save 30%, 7%, and 14% more than the oracle, respectively. However, these energy savings come at costs in performance of 24.2%, 29.1%, and 28%, respectively in Figure 13. CHILL only operates at a 1.4% energy loss, with respect to the oracle, while maintaining a 3.8% loss in performance.

H264 exhibits more streaming data behaviour, which the CHILL system can detect and power through quickly and switch to the in-order core for other phases. The CHILL system achieves 6.3% more energy efficiency than the oracle, but pays a performance penalty of 10.3%. The branching behaviour is more predictable, so the trace-based system is also able to achieve near oracle-level energy efficiency.

For hmmer, the trace-based, saturated counter, and reactive systems collectively operate at about 5% more energy efficiency than the CHILL system. This is because hmmer has a combination of more frequent CHILL phases with varying branch misprediction rates. The frequent CHILL phases map more frequently to the OoO core, which reduces energy savings. However, the trace-based system is more reactive during phases with varying branch misprediction rates on the in-order core, because its trace mechanism is based on merging backward branch PCs. This causes more unnecessary core switching in the trace-based system, which leads to a slight 1% performance drop, relative to the reactive systems (Figure 13). The saturated counter and reactive systems both have built-in mechanisms to detect branch misprediction rates (Section 4.5), and are able to obtain comparable energy efficiency. Nevertheless, the CHILL system exhibits about 23% more in performance than the other systems.
The libq benchmark experiences relatively few CHILL phases, but has separate periods of high branch misprediction rate. This combination of characteristics allows the CHILL system to find more opportunities for energy savings, with 6.1% more savings. However, the energy savings are not as high as astar because the varying branch misprediction rate causes more transitions between the cores, which is also reflected in the saturated counter and reactive schemes’ energy declines. The trace-based system operates at 65.8% energy efficiency because it is based on backward branch PCs to define phases, and high branch misprediction rates cause the system to switch cores unnecessarily. This unnecessary switching also causes a 7% performance drop (Figure 13) relative to the saturated counter and reactive systems.

6.4 Sensitivity

In order to determine the optimal epoch size for fine-grained scheduling in our benchmarks, we tested multiple epoch sizings on the oracle resource mapper. Figure 15 shows the IPC and energy savings achieved by the oracle mapper for exponentially increasing numbers of instructions. The IPC remains relatively steady after 128 instructions, and the energy savings peak at 512 instructions. As stated in Section 2, longer epoch sizes experience more uniform performance behaviour, and experience less opportunities for energy savings. We use 512 instruction sized epochs in our system to obtain the best combination of IPC and energy savings.

To find the optimal number of CCT entries, and consequently PCT entries, we tested for how often the CCT entries were actively reused for increasingly more total CCT entries. A reused CCT entry means that the entry has an active countdown initiated, and that there are active pending chains with the same branch PC as the CCT entry. This indicates that a past chain of LLLs is again experiencing a CHILL phase currently. Figure 16 shows that most benchmarks experience the most reuse at two entries. We omit astar and libq because although they experience reuse of CCT entries, they also have high data locality. This leads them to experience past CHILL phases without LLLs because the data is being reused. Most of the gains in astar and libq come from our cold start policy for pending chains described in Section 4.5. Hmmer experiences its maximum reuse of 101% with four entries, indicating that sometimes more than one pending chain is correlated to one CCT entry. To accommodate hmmer, we use five CCT and five PCT entries in our system.

7 Conclusions

High performance variance in fine-grained program phases exposes opportunities for performance and energy savings, especially on tightly-coupled heterogeneous multi-cores. A particularly important fine-grained program phase occurs during bottleneck CHILL phases. These phases are troublesome because they produce LLLs, and clog the instruction window with other dependent shadow instructions. However, the shadow instructions present opportunities for phases of high ILP and performance. We have designed a system to track and predict bottleneck CHILL phases, map the phases to suitable cores, and achieve near oracle performance and energy savings.

References


