Using an Adaptive HPC Runtime System to Reconfigure the Cache Hierarchy

Ehsan Totoni, Josep Torrellas, Laxmikant V. Kale

Department of Computer Science, University of Illinois at Urbana-Champaign, Urbana, IL 61801, USA
E-mail: {totoni2, torrella, kale}@illinois.edu

***Abstract*—**

The cache hierarchy often consumes a large portion of a processor's energy. To save energy in HPC environments, this paper proposes software-controlled reconfiguration of the cache hierarchy with an adaptive runtime system. Our approach addresses the two major limitations associated with other methods that reconfigure the caches: predicting the application's future and finding the best cache hierarchy configuration. Our approach uses formal language theory to express the application's pattern and help predict its future. Furthermore, it uses the prevalent Single Program Multiple Data (SPMD) model of HPC codes to find the best configuration in parallel quickly. Our experiments using cycle-level simulations indicate that 67% of the cache energy can be saved with only a 2.4% performance penalty on average. Moreover, we demonstrate that, for some applications, switching to a software-controlled reconfigurable streaming buffer configuration can improve performance by up to 30% and save 75% of the cache energy.

I. Introduction

Power- and energy-related issues are of growing concern in computer systems. The number of transistors on a single chip has already surpassed one billion and continues to increase. Although semiconductor processes can give us ever more transistors, thermal dissipation and broader power and energy constraints will limit their use.

These limitations have a direct impact on science and engineering applications in High-Performance Computing (HPC). The HPC community is aiming to keep the total power intake of future Exascale machines at tens of MW, whereas current systems with only 10 PetaFLOPS of performance are already consuming over 10 MW of power. Multiple innovations must be developed to dramatically reduce the total power usage of supercomputers.

The cache hierarchy has potential for many such innovations. A significant fraction of the power used by a processor chip is consumed by the cache hierarchy. For example, caches in IBM's POWER7 consume around 40% of the processor's power [1]. Yet, the caches may not be utilized equally in various Computational Science and Engineering (CSE) applications and even across different phases of a single application.

To exploit this fact, we have developed a scheme that saves energy by using the runtime system (RTS) to selectively turn off parts of the caches. Our approach takes advantage of common characteristics of workloads found in CSE. We also leverage adaptive RTSs that include an introspective component that is aware of both the current hardware status and the application.

In this paper, we characterize HPC platforms and applications and find common patterns of cache utilization. We then use these patterns to develop a novel, adaptive RTS-based scheme that automatically turns parts of the caches on or off to save energy. Our scheme also switches the cache to a streaming organization depending on the application's behavior. In this case, the runtime reconfigures the streaming parameters for best performance and energy efficiency.

Our scheme addresses major limitations associated with other methods that reconfigure the caches. It uses *persistence* and formal language theory to express the application's pattern, and the Single Program Multiple Data (SPMD) model to find the best configuration concurrently. This approach is practical since it only requires minor hardware support.

We evaluate our scheme using cycle-level simulations of a chip multiprocessor running the Mantevo mini-apps suite [2] and real applications such as NAMD [3] and MILC [4]. Our results indicate that 67% of cache energy can be saved on average, with only a slight performance penalty. We also demonstrate that adaptively switching to a reconfigurable streaming organization for the L3 cache (prefetching cache lines for detected memory-access streams) can improve both performance and energy efficiency with various tradeoffs. For example, performance can be improved by 30% while saving 75% of cache energy consumption.

The contributions of our work can be summarized as follows. We analyze the memory access characteristics of common HPC applications using the inherent properties of scientific domains and algorithms. In contrast to previous HPC characterization works [5], [6], [7], [8], we consider all the relevant aspects from algorithms to hardware. We also examine the capabilities of HPC runtime systems, and the related energy reduction possibilities in caches. Taking into consideration all the involved system components, we propose a novel cross-layer solution for adaptive cache reconfiguration. We also propose a software-controlled reconfigurable streaming scheme that can improve performance and energy efficiency for many common applications. Our proposals are highly practical since the RTS is easy to change, and the hardware complexity does not increase significantly. To the best of our knowledge, this is the first work to use HPC runtime systems to reconfigure the cache hierarchy for energy efficiency.

This paper is organized as follows: Section II and Section III discuss the necessary background and common pat-

SC14, November 16-21, 2014, New Orleans
978-1-4799-5500-8/14/$31.00 ©2014 IEEE

terns in HPC applications and architectures. Next, Section IV introduces our RTS-based scheme. Section V explains our evaluation methodology and presents the results of our scheme for cache reconfiguration. Section VI explains our reconfigurable streaming scheme and presents the results and their analysis. Section VII discusses the related work and we conclude the paper in Section VIII.

A. Background and Motivation

Cache reconfiguration (*cache tuning*) has been extensively studied, because of the high energy consumption of caches [9], [10], [11], [12], [13], [14]. Dynamic hardware-based methods need to 1) monitor the application to predict the future, and 2) find the best cache configuration effectively. Both of these stages have considerable performance and energy overhead [15], eliminating the benefits of reconfiguration. For example, the hardware could monitor some system metrics such as Instructions Per Cycle (IPC) and cache miss rate in a short interval and choose a new configuration. However, there is no guarantee that the interval is representative of the application execution, especially for HPC applications, which typically have long iteration times. In general, phase change detection is known to be challenging. Furthermore, the chosen configuration may not be the best possible, and good design space exploration heuristics are difficult to design for complex modern processors. In Section VI, we analyze a typical case where the miss rate is decreased but the performance is degraded due to different, complex factors of a modern speculative processor. A survey by Zang and Gordon-Ross [15] explores the challenges. Because of these issues, these hardware methods have not found their way to modern processors, and some recent processors, such as Angstrom [16], rely on software to reconfigure their caches.

Cache reconfiguration by the compiler has also been proposed [17], [18]. Many assumptions are made for the required footprint analysis, such as having only simple nested-loops and affine functions for array indices (only constants and loop index variables are allowed). However, large-scale HPC codes are usually more complicated. Moreover, the hardware complexities mentioned previously can prevent the compiler from choosing good configurations. We argue that the RTS can perform this task much more easily and effectively.

As an example of current practice on modern HPC machines, let us consider the simulations that were used for a recent scientific discovery at Illinois. Researchers used NAMD to simulate an HIV molecular system with 64 million atoms. Considering the maximum possible cache utilization with all the read only data (e.g. structures of atoms) and transient data (multicasts of force calculation results), only about 400 bytes per atom are needed. Therefore, the application uses only 25.6 GB of data in the working set. Note that in NAMD, the main data being updated in each iteration are the position and velocity of the atoms, which need only 48 bytes per atom combined. A typical simulation uses about 4000 Cray-XE nodes of Blue Waters (each containing two AMD Interlagos processors) with a total of 256 GB in L2 and L3 caches. Thus, more than 90% of the cache capacity was not used for the simulation. Each simulation took more than 16 days of wall clock time, which translates to a huge waste of power in the caches. Section II explains why the algorithms of this particular class of HPC applications do not need large caches.

II. HPC SYSTEMS

A. Provisioning Practices

Machines in HPC data centers are used very differently than non-HPC ones. Usually, there is no multi-programming or time-sharing of different jobs. In addition, there is no co-location of different jobs on the same nodes. Therefore, each node is dedicated to a single job at a time.

Furthermore, there is no migration of jobs across nodes. A set of nodes is dedicated to a single job for its entire execution time, which is usually much longer than the execution times of non-HPC jobs. Note that capability supercomputers usually try to run long jobs with large allocations (especially on the full machine) to facilitate new and significant scientific discoveries. On the other hand, non-HPC data centers run short and small jobs (e.g. search queries) that can be migrated using virtual machines. Thus, HPC machines are simpler to analyze and there is much more predictability and persistence in HPC data centers that can be exploited.

The processors in current supercomputers are often commodity chips. The reason is that designing and manufacturing a processor is a large investment that needs larger markets. Thus, most processors used in HPC are designed for commercial workloads in various environments, which can be very different than HPC workloads. This can result in inefficiencies of both power and performance in HPC environments. However, as we demonstrate in this paper, these can be overcome with minor support for HPC.

B. Applications

Common HPC applications are usually iterative and *persistent*, meaning that their computation and communication patterns tend to persist over time. They perform roughly the same (or very similar, at least from a memory access pattern perspective) computations and communications in each iteration. Each simulation consists of thousands to millions of these iterations (each iteration might be structured and have phases in itself, which is discussed later). For example, to simulate a bio-molecular system (e.g. in NAMD [3]), forces need to be integrated for every one (or few) femtosecond(s) of simulated time. Therefore, a one microsecond simulation of a bio-molecular system takes one million iterations. Even though the simulation is dynamic and the molecules and atoms might move across regions, the computations are roughly the same. Thus, many scientific and engineering applications follow the *principle of persistence*. This means that the computation and communication tends to persist or change slowly over time. This principle allows the RTS to predict the future of the application and has led to many successful features, such as measurement-based load balancers [19]. All the mini-apps in the suite we consider are iterative and highly persistent.

Some HPC applications (e.g. stencils and matrix-vector multiplies) are memory-bound and have lower temporal locality but higher spatial locality than other workloads. This property is studied extensively in the literature [5], [20], [7], [8]. For example, a sparse matrix vector multiply (SpMV)

kernel sweeps the matrix and vector linearly and there is a high chance of accessing neighboring values. However, if the matrix and vector inputs are larger than the largest cache, the data will not be present in the cache for the next iteration. For example, a physical domain with 100^3 grid cells per processor[1] and 240B of data per cell (for different attributes such as velocity, energy, mass, their derivatives, etc.) will occupy 230MB of memory, which should be updated in every iteration.

On the other hand, some applications have high temporal locality as well as spatial locality, but their working set (in typical execution runs) is usually much smaller than the cache hierarchy. Many Molecular Dynamics (MD) applications, such as NAMD, usually fall into this category. For example, 1000 atoms per processor with 80B of data per atom takes only 78KB of memory, which is only a small fraction of a typical Last Level Cache (LLC).

Most HPC applications follow similar memory access patterns. To study these patterns, we use the Mantevo suite's mini-apps as representative of common HPC applications. Mini-apps are simplified versions of applications that are of interest to the CSE community. They are designed to be similar to real applications from a computational perspective, and more representative than micro-benchmarks. The mini-apps of the Mantevo suite can be divided into three classes:

1) Stencil computations (CloverLeaf and MiniGhost)
2) Sparse Linear Algebra (HPCCG, MiniFE and MiniXyce)
3) Particle simulations, such as molecular dynamics (MiniMD and CoMD)

Stencil computations, which are key kernels of many structured grid applications and their PDE solvers, have limited data reuse. These kernels sweep through domain data structures that are typically much larger than caches and fill a sizable fraction of the main memory. Thus, they do not benefit from caches to the full extent. For example, Figure 1 illustrates a 5-point 2D stencil computation. In this example, the update of Point 3 uses Points 1, 2, 3, 4, and 5, which can potentially result in four cache misses. The next update, which is for Point 4 (displayed with dotted lines), can reuse some of the data of the previous update. Therefore, a memory location is reused only a few times after its first use. Similarly, consecutive updates go through the top, middle and bottom rows of the data domain in the figure. From the memory hierarchy point of view, three different address ranges are being read ("streamed") with few reuses.

In essence, these classes of applications have much more spatial locality than temporal locality. Thus, streaming buffers (or other forms of block transfers) can be more effective than typical caches for stencil computations. Streaming strategies can capture more of the available spatial locality in stencil codes to hide memory latencies. Also, spatial locality is partially captured through the cache line in caches, and a smaller cache could be just as effective.

Note that cache tiling (blocking) optimizations reduce the dimension sizes, increasing the number of reuses. However, the block size needs to be tuned and does not need to fit the whole

[1] by "processor" we mean a processor chip in this paper (not a single core).

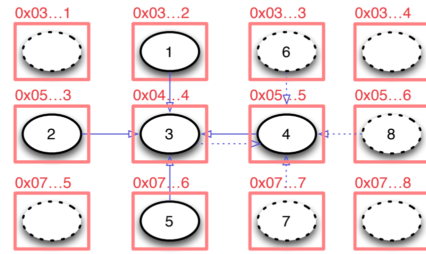


Fig. 1. 5-point 2D stencil example: boxes represent memory locations, ovals represent stencil data points, and arrows indicate data dependencies

LLC, as previous work demonstrates [21]. Furthermore, many legacy HPC codes do not incorporate these optimizations, and the required programming effort is a burden. The applications and mini-apps we evaluate do not use tiling for stencils for the same reason. Programming paradigms such as Hierarchically Tiled Arrays (HTA) [22] can alleviate this issue.

Sparse Linear Algebra computations in HPC applications also have limited cache utilization if the domain is large enough. The HPCCG, MiniFE, and MiniXyce mini-apps use sparse linear algebra methods, such as Conjugate Gradient (CG) and Generalized Minimal Residual (GMRES). Most of the execution time of these methods is spent in matrix-vector and vector-vector operations. These kernels stream data from main memory without much reuse. For example, matrix-vector multiply kernels read the matrix only once in every iteration. The vector accesses might also not have much data reuse depending on the structure of the matrix (e.g. a matrix from a regular 2D grid). In addition, if the matrix is large enough, the vector is evicted from the cache. Thus, consecutive addresses from a few address ranges are read regularly (from the memory hierarchy) for these kernels. Therefore, they have high spatial locality, similar to stencils, and the same arguments apply.

Many Molecular Dynamics (MD) and other particle interaction kernels are different than the previous categories and can have high temporal locality (as well as spatial locality). The reason is that the previous categories usually represent discretized points in the physical domain, while particle kernels represent entities. Each entity can have many interactions with other entities, while a point is fine-grained and usually interacts only with its neighbors. For example, in many particle kernels each particle (an atom in MD or a star in astrophysics) interacts with all other particles within a cut off distance. Hence, while each memory location is accessed $O(1)$ times in the other two classes of applications, it is accessed $O(n)$ times in particle applications (n is the number of particles in a cut-off), resulting in high data reuse. On the other hand, the data size for practical runs is typically smaller than caches of modern machines, especially the LLC. One reason is that the order of the computation time is roughly the square of the data size in the cut-off, making large input sizes impractical. Thus, large caches are not exploited to their full potential in many members of this class of applications either.

As we discussed earlier, cache effectiveness of common HPC applications is highly related to their per-processor working set size. Therefore, even a single application can have different cache utilization profiles depending on the input size

and the number of processors used. Thus, there is no single cache hierarchy configuration that could fit all cases, providing the highest performance and energy efficiency.

C. Runtime Systems

In HPC environments, the RTS mainly mediates the communication and provides parallel services, such as message passing in MPI. This parallel management, in addition to applications' persistence, empowers the runtime system to provide other important features such as load balancing [19], fault tolerance [23], efficient parallel I/O [24], [25], and power management [26], [27]. Therefore, an adaptive RTS orchestrates a control system [28].

Our approach is based on the management of Sequential Execution Blocks (SEBs), which we define as sequential computations between two communication calls (e.g. MPI calls). The RTS has control before and after each SEB, but it cannot usually interrupt it. These SEBs are repeated every iteration and they perform roughly the same computation (especially from a cache access perspective). For example, in most stencil codes the processors iteratively exchange the boundaries and update their values in an SEB. Thus, we try to adapt the caches to the SEB that is about to execute.

III. Cache Hierarchy

A. Cache Structure

Modern processors have multiple levels of very large caches to hide memory latency as much as possible. For example, the Intel Xeon E7-8870 [29] has 30MB of L3 cache in SRAM technology and IBM POWER8 [30] has 96MB of L3 cache in eDRAM technology. Architects try to incorporate larger caches to accelerate different workloads, while meeting the area, power, and latency budgets (e.g., it is critical in many designs to have only one cycle latency for L1 caches). Therefore, a large fraction of the silicon area is used by caches.

The cache hierarchies are designed for a diverse set of applications and hence, a fixed design might not be best for every workload. Furthermore, most supercomputers use commodity processors, which are designed for other (non-HPC) markets with different workloads. These factors result in immense waste of power and energy in supercomputers. For example, "big data" applications such as graph analysis might not have any locality because they are unstructured in their memory accesses (e.g. *pointer chasing* pattern). Thus, a level of adaptivity is needed to match the running application without too much hardware overhead.

B. Cache Power

Caches consume a large fraction of processor chips' power budget. For example, even with many advanced hardware power reduction techniques in place, caches in POWER7 consume around 40% of the total power [1]. The power consumption of caches depends on the technology, but our approach can help in most cases. SRAM technology has high leakage but is faster. On the other hand, eDRAM has much less leakage and higher capacity but needs to be refreshed [31]. Our approach can help with either technology.

Turning off ways of caches, used in our approach, can save the power consumption of various caches differently. In conventional caches, tag lookup and data access are performed in parallel for faster access. Therefore, considering that a large fraction of dynamic energy is consumed in data arrays, turning off ways of the cache saves significant dynamic energy in addition to leakage (static) energy. On the other hand, the caches that are not on the critical path of the processor can be made so that tag and data accesses are done sequentially, accessing only one way after tag lookup [32]. Thus, turning off ways can only save leakage energy in this case. This usually applies to Last Level Caches (LLC), which consume a lot of leakage energy.

C. Architectural Opportunities

Modern set-associative caches are partitioned into multiple sub-arrays for performance reasons, and only minor hardware modifications are required to turn them off. Previous work proposes to do so, through simple changes to the cache controller and the addition of a register to let software turn ways off [32].

Most recent processors (and proposals) incorporate advanced features which let the software control various aspects of the processor similar to what we need, so our proposal is practical. The Angstrom architecture [16], which has been proposed for extreme-scale computing, allows the cache size to be changed by turning off ways and banks of the caches in software. The RAPL (Running Average Power Limit) [33] interface of recent Intel processor lets the software limit the power consumption of the chip among other features. The architectural support we need for our approach seems to be much simpler than RAPL.

D. Streaming

Some related proposals focus on streaming strategies as a cache efficiency technique for scientific applications [34]. In short, streaming relies on the spatial locality of HPC applications to load more data to reduce memory access latency. This can improve both performance and energy efficiency. In this work, we propose a unique software-controlled reconfigurable streaming scheme.

Streaming schemes strive to recognize the memory accesses with simple patterns (*streams*) and prefetch them using prefetch buffers. When a memory access misses in the cache, a stream is allocated and the cache blocks are prefetched starting from the missed target. Thus, subsequent memory accesses of the stream will have the data available in the cache. This method is simple, but it is usually effective for HPC applications because of the common patterns discussed in Section II.

Since most HPC applications usually access multiple arrays in each loop (e.g. pressure and temperature at each grid point), using more than one stream is useful (i.e., *multi-way streams*). When a memory access misses in the cache (i.e. it is not prefetched by the other streams), an old stream is flushed and a new stream is allocated starting at the miss address. We assume a Least Recently Used (LRU) policy to select the stream to be deallocated.

The *depth* of the stream is an important parameter. Streams should be deep enough to hide the memory latency for the subsequent accesses, but not *too* deep. If a stream is too deep, it competes with other useful accesses, potentially delaying them. Also, it can evict useful blocks from the cache. Thus, it can waste memory bandwidth and energy. We evaluate the reconfiguration of the depth of streams in Section VI.

It is important to filter isolated references, as allocating streams for them can waste memory bandwidth and energy. This is done with a small *history buffer* that stores the addresses of recent misses. A stream is allocated only when a miss to a block occurs next to a previously missed block (e.g. a and $a + l$). This also facilitates the detection of non-unit strides that are prevalent in some HPC applications (e.g. accesses to addresses of a, $a + d$, and $a + 2d$, for $d > 1$). A stream can have unit strides or non-unit strides depending on the application's memory access pattern.

In this paper, we reconfigure the cache's (used as the streaming buffer) size and prefetch depth automatically to improve performance and energy efficiency significantly. This needs some hardware support (e.g., a small stream detection table), but it is relatively low cost. For the streaming implementation in this work, we use a previous work [34] that identifies a few streams of constant stride. However, we do not use a separate stream buffer and prefetch to the LLC instead (more details in Section VI).

IV. Reconfiguration in Adaptive Runtime Systems

In this section, we introduce our approach for automatic way reconfiguration of the cache hierarchy. First, we present our baseline approach, which is enough for many common applications. Then, we formulate the general problem of application pattern recognition to incorporate more applications.

A. Overview of Our Approach

The RTS can easily identify common iterative patterns. This can be done by monitoring communication calls (e.g., MPI_Recv, MPI_Barrier etc.) to see if they are repeated regularly with similar arguments. The time between the matching calls is the iteration time and should be reasonably consistent. Even complex HPC patterns can also be expressed using Formal Language Theory, as discussed later. Alternatively, some applications have calls to the RTS marking the end of each iteration, which are used for other purposes such as load balancing and fault tolerance. For example, calls that mark the best place for checkpointing are usually made between iterations.

Figure 2 presents the time between successive calls to MPI_Allreduce in MILC [4], which is a prominent code for Quantum Chromodynamics (QCD), running on a BlueGene/Q system. The *Allreduce* collective is called in the Multi-CG solve phases of MILC, which designates the CG steps. As can be seen, there is a clear regular pattern even for this sophisticated application. After removing the outliers, the average is 6.893 ms with a standard deviation of only 0.045 ms. In addition, the patterns for other phases of MILC are also regular (but with nearest-neighbor communication instead of collectives). Thus, even a sophisticated application such as MILC has a repetitive and regular pattern that is recognizable

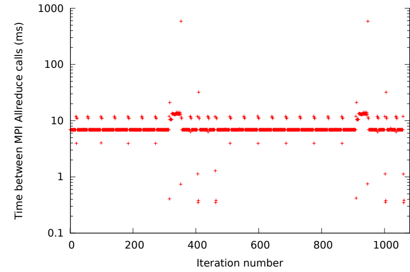


Fig. 2. Time between calls to Allreduce in MILC

by the runtime system using the the time between communication calls.

Note that, just like other runtime adaptation mechanisms such as load balancing, we ignore the initialization and some iterations in the beginning of the execution. However, since initialization is usually a small fraction of the execution time, using the best configuration is not essential.

After identifying the iterative pattern and reconfiguration units (i.e., Sequential Execution Blocks (SEBs)), the RTS should ensure the iterations and their SEBs are the same across different processors. For this purpose, the runtime gathers some *characteristic information* about the execution of each SEB on different nodes, including execution time, instruction-based samples [35], and key performance counters. Then, reconfiguration is applied if the attributes are within a threshold on all nodes. This can be accomplished by collective calls (e.g. Allreduce) that determine if the collected attributes are statistically similar (e.g. the minimum and maximum of the attributes are not too far from the average).

After finding the persistence pattern, the best cache sizes need to be found for each reconfiguration unit (SEBs or whole iteration). For example, the (2,1,2,4) configuration specifies that 2 ways need to remain active for L1D cache, 1 way for L1I cache, 2 ways for L2 cache and 4 ways for L3 cache. This is accomplished by applying and benchmarking different configurations on different processor in parallel to find the best one. We can map the configurations to sequential numbers and each processor can use its number (e.g. derived from MPI ranks) to know which configuration it needs to try. Different processors measure the execution time and energy consumption for some number of iterations and the minimum is reported by collective calls (e.g. Allreduce). The best configuration is then used on all the processors.

When the best configuration is applied, the RTS observes the execution of future iterations until the attributes become significantly different. Then, our method is invoked again to adapt to the change. Note that if the variation of the application is so high that our method could degrade performance, we switch to the default configuration (full size caches and normal policy), which is the "safest." In our experiments, we found that the runtime would not need to switch to the default configuration very often, but this is possible in the general case.

Our approach in the RTS can be summarized as follows:

1) Determine iterations (and relevant SEBs)

2) Ensure the SEBs are the same across processors
3) Run different configurations on different processors and find the best in performance and power/energy efficiency
4) Apply the best configuration to all processors
5) Observe the execution and repeat if behavior changes

Note that we depend on the fact that SEB characteristics are the same or similar on different processors. This follows from the Single Program Multiple Data (SPMD) paradigm assumed in most distributed memory parallel languages, such as MPI.

B. Generalization

Most scientific applications are structured: they can have multiple phases in each overall iteration, but these phases are also often iterative, forming a "hierarchical" iteration structure. For example, Figure 3 depicts different phases of MILC on four processors. This is a timeline diagram, where different phases (e.g domain updates with nearest neighbor communication, and CG solve) are color-coded differently. Note that the executions of four processors are stacked, but they appear very similar.

Using Formal Language Theory, the hierarchical iterative structure of an HPC application can be expressed as a *Regular Language*. We define each unique SEB as a symbol a of an alphabet Σ. Each application execution might have a different number of iterations and hence, is a *word* of the language.

Theorem. *A hierarchical iterative pattern is a regular language.*

Proof by construction: Each execution is a number of repeated iterations. Therefore, the pattern can be written as a regular expression of this form: $(a_0, a_1, ..., a_d)^*$, where each a_i is a regular expression for each (possibly iterative) component of each iteration. The regular expressions a_i can also be constructed in the same way, since each component is iterative as well. Following this procedure, in a finite number of steps, the whole regular expression can be constructed recursively. Hence, the language is regular, since it has a regular expression. ∎

The general problem of finding the application's pattern (to use for phase change detection) is a pattern recognition problem. Using our formulation, it can be modeled as a classical Formal Language Theory problem: *learning a regular language from text* [36], [37]. During the application profiling, we collect a stream of symbols that are from a regular language, and we need to infer the language.

In the profiling phase, we gather a string of symbols (*Sample S*) of the language by monitoring the SEBs. We need to infer the grammar to build a deterministic finite automaton (DFA). Recall that a DFA is a tuple $(\Sigma, Q, q_\lambda, F, \sigma)$ where Σ is a finite alphabet, Q is a finite set of states, q_λ is an initial state ($q_\lambda \in Q$), F is a set of final states ($F \subseteq Q$), and σ is a transition function ($\sigma : Q \times \Sigma \to Q$). For example, Figure 3 can be rewritten as a list of symbols: $a_0 a_1 a_2 a_3 ...$

A simple solution is to use a *prefix tree acceptor (PTA)* [38], [37]. A PTA is a tree-like DFA that has all the prefixes of the sample as states, and is *strongly consistent* with the sample, which means that it only accepts the sample[2]. Algorithm 1 demonstrates how a PTA can be built from a sample. In essence, the PTA has one state for each prefix of the sample. For example, if the sample is abc, the PTA has a state for a, ab, and abc. The transition function has only one transition per state, which goes to the state representing the next longer prefix. For example, the state for a only goes to the state for ab. Figure 4 illustrates the PTA that is built by this algorithm for sample abc.

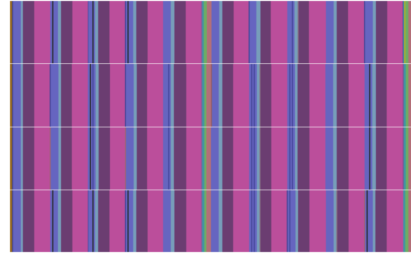


Fig. 3. Timeline view of phases of MILC: time is on x axis and four processors are stacked on y axis. Colors represent different computations. This figure illustrates the regular iterative pattern of MILC.

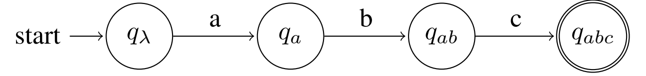


Fig. 4. PTA for sample abc.

Algorithm: Build-PTA
Input: Sample S
Output: DFA A=$(\Sigma, Q, q_\lambda, F, \sigma)$
$F \leftarrow \emptyset$;
$Q \leftarrow \{q_u : u \in PREF(S)\}$;
for $q_{u \cdot a} \in Q$ **do**
  $\sigma(q_u, a) \leftarrow q_{u \cdot a}$;
end
$F \leftarrow F \cup \{q_S\}$;

Algorithm 1: Build PTA from sample.

Learning from text by a PTA can be challenging since the number of states can grow large. However, in practice, the number of SEBs that execute in the profiling stage is small. Furthermore, the number of DFA states can be reduced easily. For example, the application might have 1000 relaxation steps followed by 1000 CG steps in each overall iteration. This translates to 2001 DFA states since there these many prefixes in the string of the iteration pattern in our formulation. To reduce this number, we combine all of the CG steps together to form only one symbol since the same SEB is repeated. This fits our purpose since similar SEBs will have the same cache configuration. In this way, our example will have only three states in its DFA. Note that there are *state merging techniques* that can be used to merge *compatible* states (please refer to the references [36], [37]). However, for practical cases, the number of states is already very small after applying our technique.

The inferred DFA (equivalent to a regular expression) will be used for the rest of the application execution by the RTS

[2]We have simplified the definitions and the algorithm for our purpose but in general, there can be multiple *positive* and *negative* samples of the language to learn from.

to predict the future of the application. In this formulation, predicting the future of the application is similar to simple pattern matching of regular expressions. For instance, when the RTS is in state a and the next state is ab, the RTS predicts that the SEB that b represents will be executed next, so it changes the configuration to the best one for b before its execution.

Using our approach, the patterns of sophisticated HPC applications can be expressed by simple regular expressions. For example, NAMD performs three force calculation steps (a), before an FFT for long range force calculations (b). Therefore, the regular expression $(a^3b)^*$. MILC's pattern illustrated in Figure 3 seems more complicated, but it can be expressed as the regular expression $((a_0a_1a_2a_3)^5b_0b_1b_2b_3)^*$ using our method.

Some HPC applications consist of multiple potentially very different regular modules, but they can be handled similarly. Although the application might seem more complicated, the resulting pattern is still a regular language due to the following lemma:

Lemma. *The concatenation of multiple regular languages is a regular language.*

Therefore, the RTS will construct a single DFA, encompassing the execution of all the modules, and our approach is applied without any change.

In some applications, the processors are divided into logical groups that perform different computations, and this *application heterogeneity* needs to be taken into account to be able to apply our scheme in these cases. For example, in a climate simulation application, some processors might simulate parts of the physical domain that is inside a storm. Therefore, they might be performing different computations than the processors that do not simulate a storm. Hence, the cache hierarchy requirements might not be the same for all of the processors. In general, programmers (mostly in the SPMD programming model) can divide the processors into logical groups to perform different computations. To incorporate these cases, when the SEBs are not the same across different processors, the RTS can run a parallel clustering algorithm that identifies similar processors based on their SEBs. For example, the processors simulating the storm might be running SEB a_2, while the others are running SEB a_1. In addition, the runtime can monitor the calls that are usually used by programmers for this purpose (such as MPI_Comm_split) as a hint. Then, the test of configurations phase of our approach is applied to each group separately. Also, each group can have a different DFA for pattern matching. Note that we do not require all the processors to have the same behavior. We simply need the pattern of each group to be regular.

In some applications, the phases can be slightly different across different overall iterations. As a hypothetical example, suppose the application has a CG phase in each overall iteration, and the number of CG iterations required for convergence can be between 995 and 1005. This is non-deterministic from the RTS point of view and can be modeled using a *Probabilistic Finite Automaton (PFA)* or similarly as a Hidden Markov Model (HMM) (please see references [39], [40]). However, this general formulation will increase the complexity of the problem and is not needed in our setting. To handle these cases, the runtime only needs to choose a "conservative" cache configuration when the next DFA state is not known. In our example, assume that the next phase needs a larger L3 than the CG phase. Therefore, the runtime chooses a cache configuration with a larger L3 for the last few iterations of CG (from iteration 995) because it does not know exactly when the next phase starts. This technique avoids performance degradation of the next phase.

The conservative cache configuration is constructed by examining the possible future states and setting the size of each cache level to the largest of the configurations. In our experiments, we found that this has negligible impact since only a small fraction of the execution time would need conservative configurations. Note that in some rare scenarios, the structure can slightly change (such as two SEBs running in switched orders in CHARM++ [41], [42]), but these variations can be handled conservatively as well. Moreover, the runtime goes to a conservative cache configuration when there were too many mistakes in the DFA's predictions. This avoids overheads for the uncommon applications that are not persistent. In depth study of these cases is beyond the scope of this paper.

C. Practical Details

The overheads of our scheme are very small compared to the overall application runtime and also can be measured and controlled easily by the RTS. The overhead of checking the configurations is negligible since it only impacts a few iterations. For many common applications, reconfiguration is done only once. For most others, the runtime reconfigurations are rare due to persistence. Note that scientific applications usually run for a long time and for many iterations. For example, according to the available data (for several months) of the BlueGene/P installation at Argonne National Lab (Intrepid), the average runtime of a job was 5176 seconds (6817s for jobs larger than 8k core jobs, which are less likely to be test runs). Slowing down a few iterations (usually in the hundreds of milliseconds range) is therefore negligible.

The dominant hardware overhead for reconfiguration is invalidation traffic ("flushing") in the caches, which is added to the software overheads (system calls, calculations, table lookups, etc.) caused by the RTS. Only the modified cache lines of inactive ways need to be flushed before turning them off, which typically takes less than a microsecond (a few thousand CPU cycles). In addition, using an experimental module in the CHARM++/AMPI system, we found the software overheads to be negligible. Therefore, the total overhead is usually on the order of microseconds, while SEBs are usually hundreds of microseconds. The programmer usually ensures that SEBs are long enough to amortize communication overheads, so SEBs are usually coarse enough for reconfiguration. Therefore even frequent reconfiguration can be practical.

Some communication calls need to be "combined" (considered as one call with no SEB in between) if they are too close together in time, since they are logically one communication step and they do not represent different SEBs. For example, in a particular phase MILC repeats the pattern illustrated here for each neighboring processor before doing the actual local computation:

```
      { MPI_Irecv()
~1ms  { MPI_Isend()
      { MPI_Wait()
      { MPI_Wait()
           ...
```

These calls need to be considered as one call, since all of them happen in a short one millisecond interval, and there is negligible computation in between.

In general, the unit of reconfiguration should be selected judiciously. At one extreme, it can be as coarse-grained as the whole iteration (and hence reconfiguration is done only once). On the other hand, it can be as fine-grained as each SEB (or even finer than that if possible).

In practical settings, there might be minor timing variations of the SEBs on some processors (e.g. due to correctable ECC errors sometimes occurring). Therefore, the RTS needs to average SEB attributes of multiple iterations to smooth out these minor temporal effects. Moreover, the SEB timings and attributes do not need to match exactly. They only need to be within a threshold.

V. Evaluation of Runtime Cache Reconfiguration

A. Methodology

In this paper, we use a diverse set of common scientific applications for evaluation of our scheme. This is important as previous work has demonstrated the importance of benchmark selection for cache access analysis [8]. Instead of micro-benchmarks, we use the Mantevo mini-app suite [2] and real applications, including NAMD [3] and MILC[3][4][4]. We have confirmed that all of these applications follow the patterns we described in Section II. Furthermore, all the processors execute the same or very similar SEBs (from the cache access point of view).

We also add an FFT benchmark to complement the molecular dynamics mini-apps, since their computation is usually simplified and includes only the time consuming short range force calculations. However, the long range force calculations can become significant depending on various parameter values. In NAMD, those forces are integrated every four timesteps using an FFT kernel. We use the NPB-FT benchmark to represent that FFT kernel.

For simplicity, we assume the MPI+OpenMP programming paradigm, which means that OpenMP is used for parallelization across each processor's cores. However, runtime systems of pure MPI programs and other paradigms can easily apply our method at the processor chip level as well.

For all the experiments of this section, we use SESC [43], which is a cycle accurate simulator. We simulate each unique SEB with different configurations, and find the wall clock time of each iteration for the whole application. The simulated system's parameters are chosen to be similar to real processors, and are presented in Table I. We use CACTI [44] for modeling the power and energy consumption of the caches. We assume that the ways of the L1 and L2 caches are activated in parallel for each access (for less latency), while only one way of the L3 cache is activated for each access, since L3 is not in the critical path of the processor. Thus, turning off the ways of the L1 and L2 will save dynamic energy, while it will only save leakage energy in the L3 cache.

[3]We use su3_rmd in the MILC collection, which is usually used for benchmarking.

[4]These two applications are used by thousands of scientists on large-scale supercomputers, and were among the three applications used for the acceptance test of Blue Waters at Illinois.

TABLE I. Simulated processor's parameters

Chip	8 Core CMP
Core	MIPS32, 4 issue out-of-order processor
Instruction L1 (L1I)	32 KB, 2 way
Data L1 (L1D)	32 KB, 4 way, WT, private.
L2	256 KB, 8 way, WB, private.
L3	16 MB, 16 banks, 16 way, WB, shared
Technology node	32 nm
Frequency	3.4 GHz

In this work, we consider the properties of the application domains for our selection of the input sizes. For example, in stencil codes each element represents a point in the physical domain and the iteration's computation is linear in the input size. Consequently, large sizes are more common and practical. On the other hand, large input sizes are less common in molecular dynamics since the force computation in each iteration is not linear in the number of atoms and molecules. Table II presents the input size per processor of each application in our experiments. These sizes are small compared to weak scaling runs that fill the node's main memory, but they are used for typical strong scaling runs. In addition, input sizes larger than the LLC usually behave similarly because of common streaming patterns discussed in Subsection II-B. We study the effect of input size more extensively in different experiments.

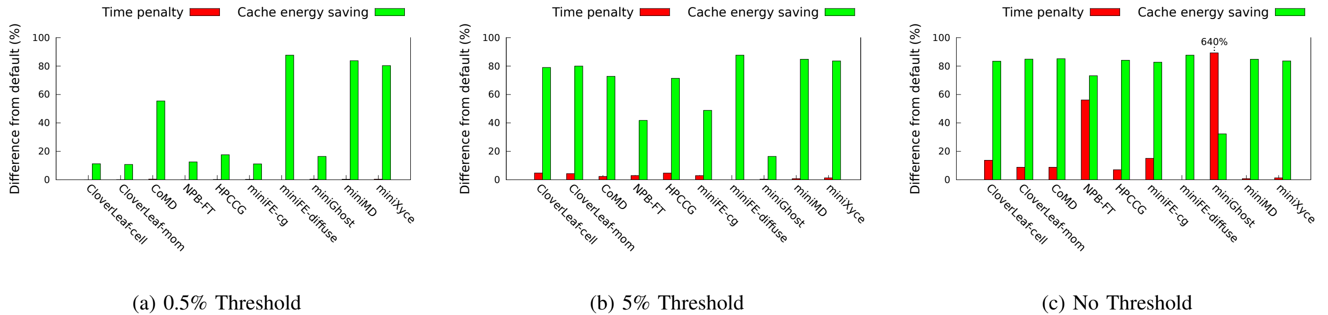
TABLE II. Application domain sizes

Mini-App	Input Domain Size per Processor
CloverLeaf	960×960 grid
CoMD	2744 boxes (including halo)
NPB-FT	$128 \times 128 \times 32$ grid
HPCCG	$60 \times 60 \times 60$ grid
miniFE	$50 \times 50 \times 50$ grid
miniGhost	$100 \times 100 \times 100$ grid
miniMD	6083 atoms (including halo)
miniXyce	602 variables

B. Results

Table III presents the cache configurations that result in the best energy efficiency, with only slight execution time penalty (0.5% penalty threshold). As can be seen, in most cases, half of the first level instruction cache and three quarters of the first level data caches were turned off for the best energy efficiency. The reason is that turning off ways of L1 caches can save a lot of energy, since they are the closest to the processor and have many more accesses. However, naive shutdown of ways of L1 caches can be detrimental, since they are critical for performance and increasing their miss rates can hurt performance significantly. In our simulation results (not presented here), some configurations with small L1 caches and not enough capacity in other caches resulted in more than one order of magnitude slow-down. Thus, the other levels need to have enough capacity to back up lower level caches, and configurations should be selected carefully.

The only configuration with multiple L1D ways enabled is for miniMD. The reason is that the working set (data structures of atoms) fits in the L1 cache. Because of the high computation per data element in molecular dynamics programs (discussed in Section II), the benefit of having them in L1 exceeds the power saving of turning off its ways.



(a) 0.5% Threshold (b) 5% Threshold (c) No Threshold

Fig. 5. Time penalty and cache energy saving of reconfiguration with different time penalty thresholds

Filtering Configurations: We try all the configurations exhaustively since there are only a few SEBs but many processors in a supercomputer. For small scale (down to one processor) runs, one could try only the configurations that are more likely to achieve better performance and energy efficiency. Table III shows that the set of high performing configurations is not diverse and only a few configurations can be the best for different applications. More investigation at the small scale is left for future work.

TABLE III. BEST CONFIGURATION FOUND WITH LOWEST ENERGY BUT WITHOUT PERFORMANCE PENALTY. FORMAT: (NUMBER OF CACHE WAYS ON)/(TOTAL NUMBER OF WAYS).

Mini-App	L1D	L1I	L2	L3
CloverLeaf-cell	1/4	1/2	2/8	16/16
CloverLeaf-mom	1/4	1/2	2/8	16/16
CoMD	1/4	1/2	2/8	8/16
NPB-FT	1/4	2/2	4/8	16/16
HPCCG	1/4	1/2	2/8	16/16
miniFE-cg	1/4	1/2	2/8	16/16
miniFE-diffuse	1/4	1/2	1/8	1/16
miniGhost	1/4	1/2	2/8	16/16
miniMD	2/4	1/2	2/8	1/16
miniXyce	1/4	1/2	4/8	1/16

Figures 5(a) to 5(c) present the execution time penalty and energy savings of different mini-apps due to reconfiguration, with different performance penalty thresholds. Note that some mini-apps have more than one significant kernel (presented separately, such as miniFE-cg), while others are simple enough to take the whole iteration as reconfiguration units. From this figure, it is evident that with negligible change in execution time (less than 0.5% performance penalty threshold, 0.2% average actual penalty), very significant cache energy savings (up to 88%) are possible. On average, about 40% of cache energy consumption can be saved by just turning off ways of caches, without a significant performance penalty.

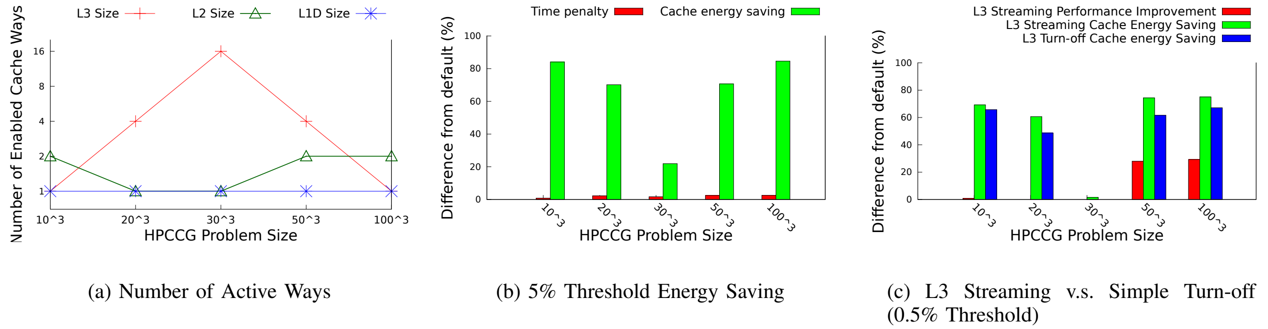
Furthermore, a small sacrifice in performance (less than 5% threshold, 2.4% average actual penalty) can result in more cache energy savings (about 67% on average). These small performance differences in the computation may not result in any performance degradation for many HPC applications because of inter-node communication. Moreover, minimizing cache energy without considering performance degradation results in more savings (about 78% average savings), but it can result in a very high penalty in some cases (6.4 times slowdown for miniGhost). This happens for miniGhost because its data fits in the L3 cache, but this method is trying to turn L3 ways off to save leakage energy. This is clearly a suboptimal decision from the energy standpoint as well, because other energy consumption sources, such as extra memory transfers, have not been considered. One should consider other energy sources if available for measurement consequently or cap the performance penalty.

Figures 6(a) to 6(c) illustrate the behavior and effectiveness of our approach for different problem sizes. Figure 6(a) illustrates that our approach initially increases the cache size (mostly L3) to incorporate the working set, which is the most energy efficient decision. However, a larger cache is not very useful for very large working set sizes and decreasing the size adaptively is the best strategy. Figure 6(b) is consistent with the previous one, demonstrating that when the working set fits in the cache, less energy savings are possible (since that energy is consumed in a useful manner, following our discussion in Section II).

Figure 6(a) also demonstrates that our algorithm sometimes prefers to have more than one way of the L2 cache active, which is consistent with the results of Table III but seems counter-intuitive in some cases. Our insight is that, especially when there are fewer L3 ways on, at least two L2 ways are needed to reduce the conflict misses in both L2 and L3. These complicated scenarios are difficult to handle by methods that do not test different configurations and only rely on system metrics.

VI. RECONFIGURABLE STREAMING

Based on the memory patterns of HPC applications, it is beneficial to use a streaming strategy for two of the three application classes we identified (Section II). Following the discussion of Section III-D, we propose an RTS-controlled reconfigurable streaming strategy. However, in our proposal, the cache organization is not changed and the system prefetches to the L3 cache instead of a specialized streaming buffer. When RTS switches to the streaming strategy, a streamer starts prefetching to the L3 cache. The streamer is a small structure which issues extra memory requests (similar to the CPU requests). Therefore, switching only involves turning the streamer on and off, which takes only a few cycles. The implementation details of the streamer hardware is similar to previous work [34]. Note that the streamer accesses are treated in the same fashion as the processor accesses (same cache line



(a) Number of Active Ways

(b) 5% Threshold Energy Saving

(c) L3 Streaming v.s. Simple Turn-off (0.5% Threshold)

Fig. 6. Reconfiguration with different input sizes

size, etc.). In this section, we use the SESC cycle-accurate simulator to evaluate our streaming approach.

There are two important parameters of the streaming strategy that need to be tuned based on the application and its input size. First, the size (number of ways) of the L3 to be used for streaming (as the streaming buffer) needs to be decided. A small streaming buffer can potentially harm performance because useful data might be evicted prematurely, increasing the cache miss rate. On the other hand, a larger than necessary streaming buffer will waste energy. Second, the best streaming depth needs to be determined carefully. Prefetching should bring enough data to the cache to hide memory latency, but too much extra data can evict useful data and waste memory bandwidth and energy. The RTS can tune these parameters dynamically. In general, choosing the streamer configuration is done in the same manner as choosing the cache configuration, and most of our previous discussions are directly applicable here as well.

The hardware implementation of software-controlled reconfigurable streaming is simple. The hardware for prefetching usually includes an adder that generates the next address to be prefetched from the previous address. The input of that adder can be exposed to software as a system register. Our approach does not add repetitive prefetch instructions (as in compiler prefetching approaches), so it avoids significant overhead.

Continuing with our HPCCG example, Figure 6(c) presents the results when the runtime only tunes the LLC cache size for streaming. The prefetch depth is fixed at four cache lines. The results demonstrate that streaming can improve performance significantly for larger input sizes of HPCCG, while saving more energy than basic reconfiguration of the L3 cache. For the 100^3 grid size, performance is improved by 30%, while saving 75% of cache energy consumption (relative to the default configuration).

Tuning the prefetch depth seems is more challenging, and the RTS is the best agent for this task. Figure 7 presents the runtime of HPCCG with different prefetch depths and cache sizes. In addition, various statistics of the system for these configurations are presented in Figure 8. As can be seen, the performance is better with more cache ways enabled, but the extra energy consumption might not be worth the slight performance increase in some cases. For example, having all 16 ways on improves performance only slightly compared to using 8 ways, but the energy cost is considerably higher as

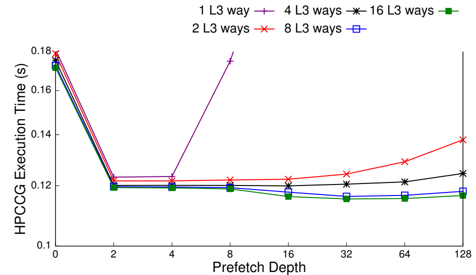


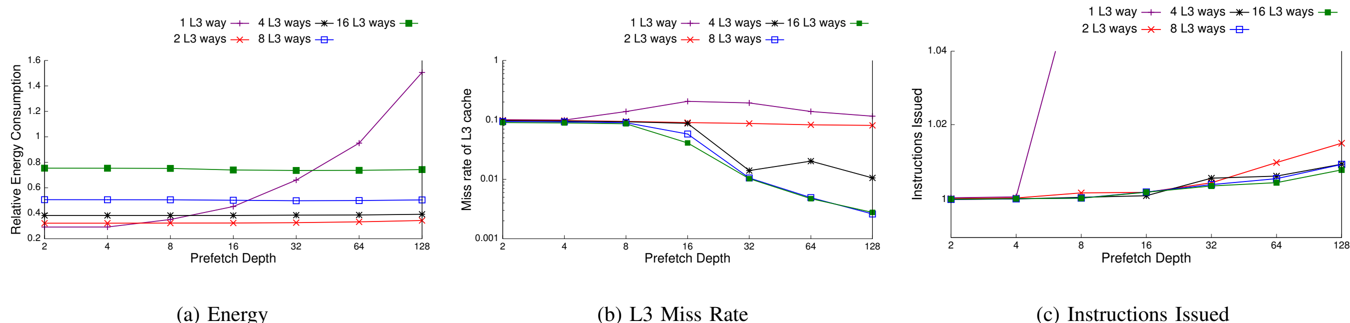
Fig. 7. Performance of different streaming configurations for HPCCG (input size 50^3)

revealed in Figure 8(a). Using eight ways of the LLC with prefetch depth of 32 seems to be a good performance-oriented tradeoff, which improves performance by 32%, while saving 50% of the cache energy. If energy is the main factor, using only one way with prefetch depth of two can save 71% of cache energy, while improving performance by 28%.

Analyzing the performance behavior of streaming strategies is complex in modern processors because of intricate cache hierarchy interactions and out-of-order/speculative execution. Figure 8(b) illustrates that in many cases, LLC miss rate decreases with very deep prefetching, but Figure 7 indicates that the performance becomes worse. More analysis reveals the reason: deeper prefetching reduces the memory delay for the mispredicted speculative paths, causing it to interfere more with the correct execution path. Figure 8(c) presents evidence for this conclusion: the number of instructions issued for the exact same computation increases with deeper prefetching. This means that the mispredicted speculative paths are making more progress and issuing more instructions, while the useful instructions committed are the same. Their excessive memory accesses evict useful data from various cache levels, harming application performance. This example demonstrates that tuning these parameters based on simple system metrics such as cache miss rate will not necessarily improve performance, and higher level software control in the RTS is needed.

VII. Related Work

The Exascale Computing Study report [45] presents energy consumption as the main challenge for future systems, with data transfer within the memory hierarchy being a large component. Other previous studies have also characterized



(a) Energy

(b) L3 Miss Rate

(c) Instructions Issued

Fig. 8. Statistics of different streaming configurations for HPCCG (input size 50^3)

scientific applications [6], [7], [8], mostly establishing that scientific applications can be incompatible with common memory hierarchies. Cicotti et al. [46] evaluate the potential of cache reconfiguration for HPC applications (without proposing a practical solution), and suggest that significant savings are possible. Our results get close to their predictions. Thus, cache hierarchy reconfiguration is promising.

Automatic cache hierarchy reconfiguration in hardware has been explored extensively [9], [10], [11], [12], [13], [14]. A survey by Zang and Gordon-Ross [15] summarizes the literature on cache adaptation. However, it is hard to predict application's phase changes and behavior in hardware and the hardware's "window" might be too small to capture the whole iteration. Also, predicting the best configuration in hardware is difficult, and incorrect hardware reconfiguration might result in extreme application performance slow-down. Moreover, automatic cache reconfiguration in hardware makes the hardware even more complicated and might also increase the energy consumption (due to additional structures, tables, etc.).

Compiler directed cache reconfiguration has also been explored [17], [18]. However, the compiler's analysis is usually limited because of the lack of runtime information. For example, array indices can be complicated in HPC application, inhibiting the required analysis. Thus, the RTS is the best agent to drive the cache reconfiguration. To the best of our knowledge, our work is the first to introduce a RTS-based adaptive cache reconfiguration in the of context HPC systems.

VIII. Conclusion

Caches consume a large fraction of a processor's power, but a fixed cache configuration does not fit every application. We exploit the regular structure of HPC applications and the partitioned structure of caches to reconfigure the caches (turn on/off ways of the cache) in the RTS, and save a large fraction of cache energy. The RTS is the best agent to direct the reconfiguration, since it can recognize the application's pattern easily (as we showed using formal language theory), without programming effort or hardware implementation overheads. Using the SESC cycle-level simulator, we demonstrated that 67% of the cache energy is saved on average, while incurring only a 2.4% penalty in sequential computation. Assuming that 70% of the total power of an HPC system is consumed by its processors, and that 40% of each processor's power goes to its caches, 19% of the total power is saved using our approach. This power can be used to turn on more compute nodes and further improve performance for over-provisioned systems. Moreover, we established that the change of cache strategy to reconfigurable streaming can save up to 75% of the cache energy and also improve performance by 30% in some cases.

Acknowledgment

This work was supported in part by Intel under the Illinois-Intel Parallelism Center (I2PC).

References

[1] V. Zyuban, J. Friedrich, C. J. Gonzalez, R. Rao, M. D. Brown, M. Ziegler, H. Jacobson, S. Islam, S. Chu, P. Kartschoke, G. Fiorenza, M. Boersma, and J. Culp, "Power optimization methodology for the IBM POWER7 microprocessor," *IBM Journal of Research and Development*, vol. 55, no. 3, 2011.

[2] M. A. Heroux, D. W. Doerfler, P. S. Crozier, J. M. Willenbring, H. C. Edwards, A. Williams, M. Rajan, E. R. Keiter, H. K. Thornquist, and R. W. Numrich, "Improving performance via mini-applications," Sandia National Laboratories, Tech. Rep. SAND2009-5574, 2009.

[3] J. C. Phillips, R. Braun, W. Wang, J. Gumbart, E. Tajkhorshid, E. Villa, C. Chipot, R. D. Skeel, L. Kalé, and K. Schulten, "Scalable molecular dynamics with NAMD," *Journal of Computational Chemistry*, vol. 26, no. 16, pp. 1781–1802, 2005.

[4] (2013) MIMD Lattice Computation (MILC) Collaboration Home Page. [Online]. Available: http://www.physics.indiana.edu/~sg/milc.html

[5] M. M. Tikir, L. Carrington, E. Strohmaier, and A. Snavely, "A genetic algorithms approach to modeling the performance of memory-bound computations," in *Proceedings of the 2007 ACM/IEEE Conference on Supercomputing (SC'07)*, 2007.

[6] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov, "Characteristics of workloads used in high performance and technical computing," in *Proceedings of the 21st Annual International Conference on Supercomputing (ICS'07)*, 2007.

[7] J. Weinberg, M. O. McCracken, E. Strohmaier, and A. Snavely, "Quantifying locality in the memory access patterns of HPC applications," in *Proceedings of the 2005 ACM/IEEE Conference on Supercomputing (SC'05)*, 2005.

[8] R. Murphy and P. Kogge, "On the memory access patterns of supercomputer applications: Benchmark selection and its implications," *Computers, IEEE Transactions on*, vol. 56, no. 7, pp. 937–945, 2007.

[9] R. Balasubramonian, D. Albonesi, A. Buyuktosunoglu, and S. Dwarkadas, "Memory hierarchy reconfiguration for energy and performance in general-purpose processor architectures," in *ACM/IEEE International Symposium on Microarchitecture (MICRO 33)*, 2000.

[10] A. Dhodapkar and J. Smith, "Managing multi-configuration hardware via dynamic working set analysis," in *International Symposium on Computer Architecture (ISCA)*, 2002.

[11] A. Gordon-Ross, J. Lau, and B. Calder, "Phase-based cache reconfiguration for a highly-configurable two-level cache hierarchy," in *Proceedings of the 18th ACM Great Lakes Symposium on VLSI (GLSVLSI '08)*, 2008.

[12] P. Ranganathan, S. Adve, and N. Jouppi, "Reconfigurable caches and their application to media processing," in *International Symposium on Computer Architecture (ISCA)*, 2000.

[13] S. Mittal, Y. Cao, and Z. Zhang, "Master: A multicore cache energy-saving technique using dynamic cache reconfiguration," *Very Large Scale Integration (VLSI) Systems, IEEE Transactions on*, vol. PP, no. 99, 2013.

[14] S. Srikantaiah, E. Kultursay, T. Zhang, M. Kandemir, M. Irwin, and Y. Xie, "Morphcache: A reconfigurable adaptive multi-level cache hierarchy," in *High Performance Computer Architecture (HPCA)*, 2011.

[15] W. Zang and A. Gordon-Ross, "A survey on cache tuning from a power/energy perspective," *ACM Comput. Surv.*, vol. 45, no. 3, pp. 32:1–32:49, Jul. 2013.

[16] H. Hoffmann, J. Holt, G. Kurian, E. Lau, M. Maggio, J. Miller, S. Neuman, M. Sinangil, Y. Sinangil, A. Agarwal, A. Chandrakasan, and S. Devadas, "Self-aware computing in the Angstrom processor," in *Design Automation Conference (DAC)*, 2012.

[17] J. Hu, M. Kandemir, N. Vijaykrishnan, and M. J. Irwin, "Analyzing data reuse for cache reconfiguration," *ACM Trans. Embed. Comput. Syst.*, vol. 4, no. 4, pp. 851–876, Nov. 2005.

[18] S. Tavarageri and P. Sadayappan, "A compiler analysis to determine useful cache size for energy efficiency," in *Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW)*, 2013.

[19] G. Zheng, A. Bhatele, E. Meneses, and L. V. Kale, "Periodic Hierarchical Load Balancing for Large Supercomputers," *International Journal of High Performance Computing Applications (IJHPCA)*, March 2011.

[20] R. Cheveresan, M. Ramsay, C. Feucht, and I. Sharapov, "Characteristics of workloads used in high performance and technical computing," in *Proceedings of the 21st Annual International Conference on Supercomputing (ICS'07)*, 2007.

[21] K. Datta, M. Murphy, V. Volkov, S. Williams, J. Carter, L. Oliker, D. Patterson, J. Shalf, and K. Yelick, "Stencil computation optimization and auto-tuning on state-of-the-art multicore architectures," in *Proceedings of the 2008 ACM/IEEE Conference on Supercomputing (SC'08)*, 2008.

[22] G. Bikshandi, J. Guo, D. Hoeflinger, G. Almasi, B. B. Fraguela, M. J. Garzarán, D. Padua, and C. Von Praun, "Programming for parallelism and locality with hierarchically tiled arrays," in *Proceedings of the eleventh ACM SIGPLAN symposium on Principles and practice of parallel programming*. ACM, 2006, pp. 48–57.

[23] E. Meneses, "Scalable message-logging techniques for effective fault tolerance in HPC applications," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2013, http://charm.cs.uiuc.edu/papers/13-17/.

[24] P. Corbett, D. Feitelson, S. Fineberg, Y. Hsu, B. Nitzberg, J.-P. Prost, M. Snirt, B. Traversat, and P. Wong, "Overview of the MPI-IO Parallel I/O interface," in *Input/Output in Parallel and Distributed Computer Systems*. Springer US, 1996, vol. 362, pp. 127–146.

[25] B. Behzad, H. V. T. Luu, J. Huchette, S. Byna, Prabhat, R. Aydt, Q. Koziol, and M. Snir, "Taming parallel I/O complexity with auto-tuning," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, 2013.

[26] O. Sarood, P. Miller, E. Totoni, and L. V. Kale, "'Cool' Load Balancing for High Performance Computing Data Centers," in *IEEE Transactions on Computer - SI (Energy Efficient Computing)*, September 2012.

[27] O. Sarood, A. Langer, A. Gupta, and L. V. Kale, "Maximizing throughput of overprovisioned HPC data centers under a strict power budget," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, ser. SC '14. New York, NY, USA: ACM, 2014.

[28] I. Dooley, "Intelligent runtime tuning of parallel applications with control points," Ph.D. dissertation, Dept. of Computer Science, University of Illinois, 2010, http://charm.cs.uiuc.edu/papers/DooleyPhDThesis10.shtml.

[29] (2014) Intel Product Information. [Online]. Available: http://ark.intel.com/

[30] (2014) IBM Product Information. [Online]. Available: http://www.ibm.com/

[31] M.-T. Chang, P. Rosenfeld, S.-L. Lu, and B. Jacob, "Technology comparison for large last-level caches (L3Cs): Low-leakage SRAM, low write-energy STT-RAM, and refresh-optimized eDRAM," in *High Performance Computer Architecture (HPCA)*, 2013.

[32] D. H. Albonesi, "Selective cache ways: On-demand cache resource allocation," in *Proceedings of the 32Nd Annual ACM/IEEE International Symposium on Microarchitecture (MICRO 32)*, 1999.

[33] I. Corp., "Intel 64 and IA-32 architectures software developer manual," 2013.

[34] S. Palacharla and R. E. Kessler, "Evaluating stream buffers as a secondary cache replacement," in *ACM SIGARCH Computer Architecture News*, vol. 22, no. 2. IEEE Computer Society Press, 1994, pp. 24–33.

[35] X. Liu and J. Mellor-Crummey, "A data-centric profiler for parallel programs," in *Proceedings of International Conference for High Performance Computing, Networking, Storage and Analysis (SC'13)*, 2013.

[36] E. M. Gold, "Language identification in the limit," *Information and control*, vol. 10, no. 5, pp. 447–474, 1967.

[37] C. De la Higuera, *Grammatical inference: learning automata and grammars*. Cambridge University Press, 2010.

[38] H. Fernau, "Learning tree languages from text," in *Computational Learning Theory*. Springer, 2002, pp. 153–168.

[39] E. Vidal, F. Thollard, C. De La Higuera, F. Casacuberta, and R. C. Carrasco, "Probabilistic finite-state machines," *Pattern Analysis and Machine Intelligence, IEEE Transactions on*, vol. 27, no. 7, pp. 1013–1025, 2005.

[40] S. R. Eddy, "Hidden markov models," *Current opinion in structural biology*, vol. 6, no. 3, pp. 361–365, 1996.

[41] B. Acun, A. Gupta, N. Jain, A. Langer, H. Menon, E. Mikida, X. Ni, M. Robson, Y. Sun, E. Totoni, L. Wesolowski, and L. Kale, "Parallel Programming with Migratable Objects: Charm++ in Practice," in *Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis (SC'14)*. New York, NY, USA: ACM, 2014.

[42] L. Kale, A. Arya, N. Jain, A. Langer, J. Lifflander, H. Menon, X. Ni, Y. Sun, E. Totoni, R. Venkataraman, and L. Wesolowski, "Migratable objects + active messages + adaptive runtime = productivity + performance: A submission to 2012 HPC class II challenge," Parallel Programming Laboratory, Tech. Rep. 12-47, November 2012.

[43] J. Renau *et al.*, "SESC: SuperESCalar simulator," 2005.

[44] "CACTI: an integrated cache and memory access time, cycle time, area, leakage, and dynamic power model," 2013. [Online]. Available: http://http://www.hpl.hp.com/research/cacti/

[45] K. Bergman, S. Borkar, D. Campbell, W. Carlson, W. Dally, M. Denneau, P. Franzon, W. Harrod, K. Hill, J. Hiller *et al.*, "Exascale computing study: Technology challenges in achieving exascale systems," *Defense Advanced Research Projects Agency Information Processing Techniques Office (DARPA IPTO), Tech. Rep*, 2008.

[46] P. Cicotti, L. Carrington, and A. Chien, "Toward application-specific memory reconfiguration for energy efficiency," in *Workshop on Energy Efficient Supercomputing (E2SC'13)*, 2013.