DOE Award #: DE-FC02-10ER25992/DE-SC0005512
Recipient Institution: University of Illinois at Urbana-Champaign

Project Title: Thrifty: An Exascale Architecture for Energy Proportional Computing

PI: Josep Torrellas (torrellas@cs.uiuc.edu)

Date of the Report: March 16, 2013
Period Covered: Semi-Annual Progress Report (9/1/12 - 2/28/13)

1. Introduction

The objective of this project is to design a novel exascale architecture called Thrifty that addresses the challenges of power/energy efficiency, resiliency, and performance in exascale systems. The project includes work on computer architecture (Josep Torrellas from University of Illinois), compilation (Daniel Quinlan from Lawrence Livermore National Laboratory), runtime and applications (Laura Carrington from University of California San Diego), and circuits (Wilfred Pinfold from Intel Corporation). More specifically, the aims of this project are:

Power/Energy Efficiency: Attain major efficiency gains over current computer systems by developing:

• Novel circuits and architecture technologies for Near-Threshold Voltage Computing (NTC) and fine-grain management of static and dynamic power.
• A power/energy-aware compiler for Fortran and C based on ROSE that uses static and dynamic code analysis to drive the power-management hardware and auto-tunes code for power/energy efficiency.
•Application models for efficient energy-proportional computing and a means to insert power-management pragmas in the application.

Resiliency: Reduce the waste in computing due to faults and recovery in high-end systems by developing:

• Novel circuit technologies for high resiliency at low supply voltage.
• New circuits and architecture technologies for efficient error detection and tolerance.
• A novel architectural scheme for energy-efficient, diskless, scalable checkpointing.
• Applications and compiler support to drive the novel checkpointing scheme.

Performance: Attain major performance increases over current machines by:

•Proposing a novel architecture (Thrifty) with many performance-enhancing features, including primitives for fine-grain synchronization and communication.
• Developing a compiler that efficiently drives the performance features of Thrifty and auto-tunes programs for performance.
• Identifying application Idioms and mapping them efficiently on Thrifty, using them to insert tuned libraries, and performing novel thread scheduling.

In this report, we focus on the progress during the period and the plans for next 6-months.

2. Progress During the Period

We describe the progress in power/energy efficiency, performance, and resiliency.

2.1. Power/Energy Efficiency

2.1.1. Energy Management Framework that Combines Architecture, Compilation and User Annotations

We have developed an Energy Management Framework composed of an API that enables the compiler or the programmer to change chip configuration parameters. The goal is to execute in a more energy-efficient manner. The API consists of library calls to:

1. Select the Voltage and Frequency Bin for each of the processors and for each of the L2 caches in the chip.
2. Turn-off (power-gate) and turn-on various resources in the chip. They include individual processors, caches, ways of associative caches, and functional units.

The Thrifty hardware architecture is designed with ability to change the voltage and frequency bins of individual components, as well as to power-gate individual components of the hardware.

The ROSE compiler has been extended to be able to analyze the code and, based on that, call the API to configure the architecture in the most energy-efficient mode. In particular, the ROSE compiler:

1. Detects serial and parallel sections in OpenMP programs. Before the serial section starts, it calls the API to power-off all the processors that will remain idle (and their caches). Before the parallel section starts, it calls the API to power-on all the processors and caches.
2. Detects at the source code sections of the code that use different functional unit types. For example, it detects the use of floating point and integer functional units. When such units are not used, it power-gates them.

The programmer is also allowed to call the API based on his/her knowledge of the program.

2.1.2. Energy-Efficient Thrifty Chip Organization for Near-Threshold Computing (NTC)

While NTC is a promising approach to push back the many-core power wall, it suffers from a high sensitivity to process variations. To cope with variations at NTC, one approach is to use multiple on-chip voltage (Vdd) domains supported by on-chip voltage regulators. As part of the Thirfty work, we have examined the types of power and energy efficiency issues that appear when we consider multiple voltage domains in such a chip.

There are several effects that challenge the cost-effectiveness of multiple on-chip Vdd domains in a power-conscious environment such as a future NTC chip. The first one is the power loss in conventional on-chip Vdd regulators. Having Vdd regulators on chip is likely the only realistic way to support many domains — utilizing many off-chip regulators is too expensive. Unfortunately, conventional regulators have typical power efficiencies of only 70-90%. Another issue is the likely need to increase the Vdd guard-band in the smaller Vdd domains, to tolerate more accentuated dynamic Vdd droops. Such deeper droops may be induced by the lower capacitance of a domain, compared to a large chip with a single Vdd domain. Finally, there is the practical aspect that the low-power operation of NTC chips will result in chips with many cores. Attempting to control Vdd on a per-core basis is very expensive in hardware. Therefore, it is likely that each Vdd domain will include several cores. The differences between the cores in the same Vdd domain, induced by the high process-variation parameters, will lead to a Vdd setting per domain that is suboptimal for individual cores — hence missing out on part of the potential gains.

As an example, we model an 11nm NTV chip with 288 cores. The chip is organized in clusters. A cluster has 8 cores (each with a per-core private memory) and a cluster memory. The technology parameters are derived from ITRS and from projected trends from industry. The nominal values of Vdd and f are 0.55 V and 1.0 GHz.

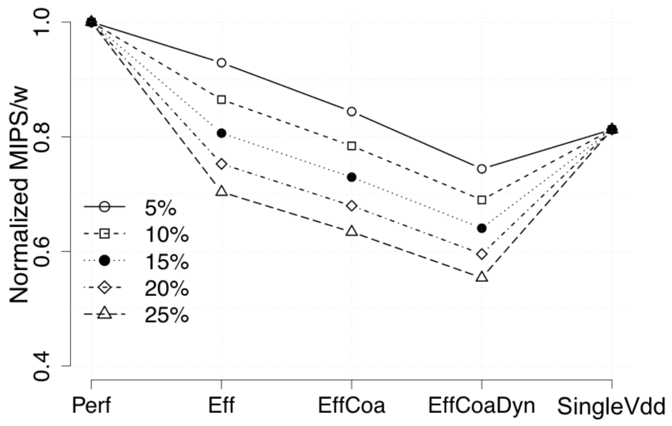


Figure 1: Comparing the energy efficiency of different environments.

Figure 1 compares the MIPS/watt of the 288-core NTC chip for the different environments. The workload consists of combinations of PARSEC applications. In the plot, we perform a sensitivity analysis of different power inefficiencies for the Vdd regulators (5%, 10%, 15%, 20%, and 25%). We compare a perfect environment with per-cluster Vdd and f domains (Perf), and then progressively add inefficiencies, such as power loss in the regulator (Eff), coarse-grain Vdd domains of four clusters per domain (EffCoa), and a 5% higher Vdd guard-band to tolerate higher dynamic Vdd droops (EffCoaDyn). We also compare to a design with a single Vdd domain (SingleVdd). We see that the inefficiencies of the on-chip Vdd regulators reduce the MIPS/w significantly. This work suggest that it is important to use Vdd regulators with high energy efficiency (preferably 95%). This work appeared in the International Symposium on High-Performance Computer Architecture (HPCA) in February 2013 [1].

2.1.3. Intelligent Refresh of On-Chip Memory Hierarchies

As the Thifty chip uses dynamic energy ever more efficiently, static power consumption becomes a major concern. In particular, leakage in on-chip memory modules contributes substantially to the chip's power draw. This is unfortunate given that, intuitively, the large multi-level cache hierarchy of a many-core is likely to contain a lot of useless data.

An effective way to reduce this problem is to use a low leakage technology such as embedded DRAM (eDRAM). However, such systems require refresh. In this work, we examine the opportunity of minimizing on-chip memory power by intelligently refreshing an eDRAM cache hierarchy (On-chip L2 and L3). We propose Refrint, a simple approach to perform fine-grained, intelligent refresh of eDRAM multiprocessor cache hierarchies. We introduce the Refrint algorithms and the micro-architecture support. We evaluate Refrint in a simulated many-core running 16-threaded parallel applications. Compared to an SRAM-only cache hierarchy, Refrint reduces the energy in the cache hierarchy by an average of 50-60%, depending on the eDRAM retention time.

The system works by saving refreshes for *Hot cache lines* and for *Cold cache lines*. Hot cache lines are those that are accessed frequently. Since an access automatically produces a refresh, Refrint saves refreshes on Hot lines by recording when they were last accessed, and not attempt a new refresh until a full retention period later. Cold lines are those that are accessed rarely. For these, Refrint saves refreshes by not refreshing them after they have not been accessed for a certain period. After a certain grace period, dirty lines are written back to the lower level memory, while clean lines are simply discarded.

We experiment with a number of refresh policies, which refresh different types of data (valid, dirty, etc.) for different periods of time before displacing it from the cache. We compare four data-based policies: *All* refreshes every cache line, irrespective of whether it is valid or not. *Valid* and *Dirty* policies always refresh Valid and Dirty cache lines, respectively. The Write Back (*WB*) policy is associated with tuple (n,m). This policy refreshes a Dirty line n times before writing it back and changing its state to Valid Clean. A Valid Clean line is refreshed m times before eviction.

In our experimental results, Refrint reduces the energy in the cache hierarchy by 60-50%. Also, of the energy remaining, the contribution of refreshes is small. Finally, it can also be shown that Refrint early invalidation and write back of lines did not increase the execution time of the applications noticeably. This work appeared in the International Symposium on High-Performance Computer Architecture (HPCA) in February 2013 [3].

2.1.4. Chip Architecture Designed from the Ground Up for Energy Efficiency

As part of this work, we worked with Intel designers to flesh out a manycore chip architecture that would fulfill Thifty's requirements for energy efficiency. We designed an architecture that is built from the ground up for energy efficiency. All of the layers of the computing stack are co-designed to consume the minimum possible energy, accepting the cost of limited compatibility with previous operating systems and applications.

The chip is intended to operate at near-threshold supply voltages. At such voltages, within-die parameter variations are expected to be significant. Consequently, much thought has been put into understanding the likely parameter variations [2], and on designing circuits and organizations to tolerate them in an energy-efficient manner. In addition, Thirfty has widespread clock and power gating in processors, memory modules, and networks.

To provide the parallelism required to achieve extreme-scale performance at the low clock rates that near-threshold voltages allow, Thrifty's processor chip includes a large number of relatively-simple cores. One likely future design is a system containing a small number of large cores optimized for ILP and a large number of simple cores, in order to provide both good performance on sequential sections of code and high parallelism on parallel regions. To exploit locality, cores are organized into groups, where each group contains a set of processors and local memories connected by an energy-optimized network.

Thrifty's memory system is designed to maximize software's ability to control data placement and movement, in the belief that this will minimize energy consumption at the cost of placing additional responsibility on the software. We provide a single, shared, address space across an entire Thrifty machine. Instead of a hardware-coherent cache hierarchy, our on-chip memory consists of a hierarchy of scratchpads and software-managed incoherent caches, and we provide a set of block transfers to minimize the cost of data movement. Our off-chip memory is implemented using stacked DRAMs with an energy-optimized interface, significantly reducing the energy consumed per bit transferred.

The on-chip network is designed with wide links to reduce latencies, but its components are power-gated when unused. In addition to the data network, we provide a network for barriers and reductions/broadcasts. This network reduces the latency and energy cost of synchronization and collective operations, both through specialized hardware and by making it easier for cores to clock- or power-gate themselves while waiting for a synchronization or a collective operation to complete.

This work appeared in the International Symposium on High-Performance Computer Architecture (HPCA) in February 2013 [3].

2.2. Performance

2.2.1. Software-Managed Cache (SMC) Framework that Combines Architecture, Compilation and User Annotations

We have developed a Software-Managed Cache (SMC) Framework composed of an API that enables the compiler or the programmer to manage the coherence of a multiprocessor cache hierarchy in software. The goal is to execute in a more energy-efficient manner and still retain high performance.

With SMCs, as a processor references a variable, the memory line containing the variable is automatically brought from memory or from the lower layers of the cache hierarchy (e.g., L3) and copied into the processor's local cache. However, caches are not coherent, which means that the processor simply gets the value that is currently in memory (or in the first level of the cache hierarchy that it finds it in). If a processor P2 wants a value that another processor P1 wrote, then P1 has to first explicitly *write back* the variable to memory, and then P2 has to *invalidate* its own cached copy before reading the variable.

We have developed a special ISA that the programmer uses to make sure the correct data is accessed. The ISA includes: 1) write-back of a word (or a range of addresses) from the cache to memory (or shared caches), 2) invalidation of a cached word (or range of addresses), 3) write-back and invalidation of a word (or range of addresses), and 4) cache-bypassing loads and stores. When a processor reads a variable and a line is brought from memory because the word was invalid in the cache, other valid words word, the whole cache line is brought into the cache. When it writes back a line, only dirty words of the line are written back to memory in a writeback, reducing the traffic significantly.

In collaboration with Professor Sadayappan from Ohio State University, we have developed a compiler algorithm using exact polyhedral dependence analysis to optimize coherence instructions for affine computations. The developed approach inserts coherence primitives – invalidate and write-back instructions – at a coarse granularity and combines invalidations and write-backs of a number of words together, to reduce both the frequency of coherence operations and the volume of words moved between private caches and the shared level cache or main memory. The experimental evaluations over a number of benchmarks show that the developed system is effective both for energy and performance metrics.

We also find that it is best to write applications for SMC from scratch, rather than simply taking existing codes and translating them. By writing from scratch, the programmer can take advantage of the data that can remain in the caches across barriers (or synchronizations) because the same processor will use it next. There is no need to move such data. In addition, data that will not be used by anyone any more can be discarded silently. This work is submitted for publication [12].

2.2.2. Potential Performance and Energy Benefits of Scratchpad Memories

Moving data causes significant losses in energy and performance in a modern supercomputer. To make the systems of today higher performance and more energy efficient, and to bring exascale computing closer to the realm of possibilities, data motion must be made more efficient. Because the motion of each bit throughout the memory hierarchy has a large energy and performance cost, energy efficiency will improve if we can ensure that only the bits absolutely necessary for the computation are moved through the hierarchy. Toward reaching that end, we explored the possible benefits of using a software managed scratchpad memory for HPC applications. Our goal is to observe how data movement (and the associated energy costs) changes when we utilize software-managed scratchpad memory (SPM) instead of the traditional hardware-coherent caches.

Using an approximate but plausible model for the behavior of SPM, we show via memory simulation tools that HPC applications can benefit from hardware containing both scratchpad and traditional cache memory in order to move an average of 39% fewer bits to and from main memory, with a maximum improvement of 69%. This work appeared in [13].

2.2.3. Fine Grain Synchronization with Full/Empty Bits

In Thrifty, we want to provide efficient atomic operations. Therefore, we have explored the impact of supporting Full/Empty bits. There are three options that we considered. One approach is a conventional design. There is no special hardware support for Full/Empty bit. Instead, we rely on the well-known Compare&Swap instruction and its inverse contrast&Swap instruction, and have a special software value to signify “empty”. The Empty value should be a value that the application will not use. For example, it can be the NaN code for floating-point locations. When the location is empty, we store the “empty” value there.

A second approach is to use simple hardware support. This can be done with a special instruction that performs a *dual* compare and swap (DCAS). The idea is to be able to lock two locations atomically. One them can act as the Full/Empty bit, and the other is the location to read/write. If any of the accesses to the two locations fails, the instruction continues to spin. The spinning is done over the network.

The final approach is to have advanced hardware support for Full/Empty bits. This design is like the previous one, but there is no spinning in the network. Instead, any requests that fail are enqueued in hardware in a hardware queue in the on-chip memory. When the owner processor releases the lock, then the next processor in line is notified. This notification acts as a response to the initial request. This solution requires more advanced hardware.

We model all these options in the simulator. The results are presented in [7]. As an example, we show data for the Laplace solver. The 1D Laplace solver uses a finite difference method to achieve numerical approximation of the equation, whose pseudo-code is shown in Figure 2. In this kernel, at each iteration, every position of a 1D array is updated with a value function of its left and right neighbors, which is computed from the previous iteration. All the elements of the array need to be updated before the next iteration starts. A conventional implementation partitions the 1D array among threads and, to enforce the producer-consumer relationship, it uses a global barrier. This barrier forces each thread to wait for all others to complete the current iteration before starting the next one. From the point of view of a thread, however, it would only need to wait for its two neighbor threads to supply the data at the border of its partition in order to continue its own computation, as shown in Figure 3. A thread only needs to synchronize the data accesses on the edge of its dataset. We do this with Full/Empty bits. Overall, this application highlights the issues with coarse-grained barriers - there is very little synchronization required in this program (only the data at the edge of a thread's chunk of data must be shared), but barriers enforce a large overhead.

```
for ( i = 0 ; i < ITERATIONS ; ++i ) {
    for ( j = 1 ; j < SIZE -1 ; ++j )
        xnew[j] = 0.5 * (x[j-1] + x[j+1] + b[j]);
    for ( j = 1; j < SIZE -1 ; ++j )
        x[j] = xnew[j];
}
```

Figure 2: Simple pseudo-code for the 1D Laplace solver.

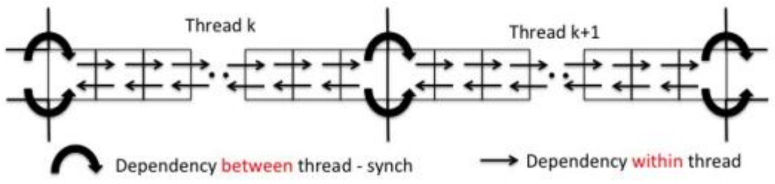


Figure 3: Data dependency and synchronization in 1D Laplace solver.

Figure 4 compares the execution time of the kernel with the conventional implementation using barriers (Baseline) and the three different hardware supports: conventional design (Single CAS), simple hardware support (Dual CAS) and advanced hardware support (Hardware). We vary the number of threads from 4 to 16.

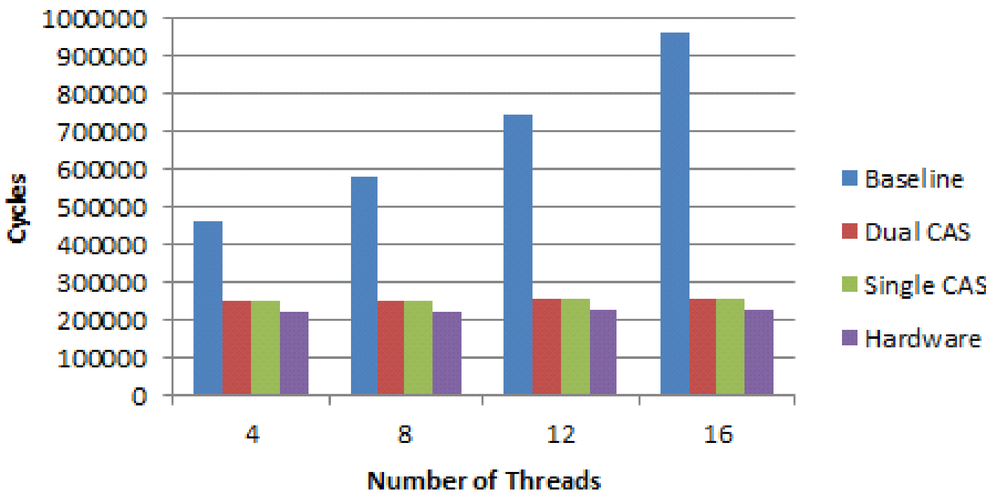


Figure 4: Execution time of the 1D Laplace solver for different synchronization supports.

We can see that having fine grain hardware support is the best choice, both in total execution time and in scalability with the number of processors. With the global barrier, the code is slow and its scalability is poor. We also see that any of the three fine-grained methods does well. This is a typical result: the design using a single CAS does nearly as well and the other, more costly approaches.

2.2.4 Illusionist: Transforming Lightweight Cores into Aggressive Cores on Demand

Power dissipation limits combined with increased silicon integration have led microprocessor vendors to design chip multiprocessors with asymmetric cores. These asymmetric multicores provide a small number of high-performance (aggressive) cores that can accelerate specific threads. However, threads are only accelerated when they can be mapped to an aggressive core, which are restricted in number due to power and thermal budgets of the chip.

Rather than using the aggressive cores to accelerate threads, we argue that the aggressive cores can have a multiplicative impact on single-thread performance by accelerating a large number of lightweight cores and providing an illusion of a chip full of aggressive cores. Specifically, we propose an adaptive asymmetric multicore, *Illusionist*, that can dynamically boost the system throughput and get a higher single-thread performance across the chip. To accelerate the performance of many lightweight cores, those few aggressive cores run all the threads that are running on the lightweight cores and generate execution hints. These hints are then used to accelerate the execution of the lightweight cores. However, the hardware resources of the aggressive core are not large enough to allow the simultaneous execution of a large number of threads. To overcome this hurdle, Illusionist performs aggressive dynamic program distillation to execute small, critical segments of each lightweight-core thread. A combination of dynamic code removal and phase-based pruning distill programs to a tiny fraction of their original contents. Experiments demonstrate that Illusionist achieves 35% higher single thread performance for all the threads running on the system, compared to a multicore with all lightweight cores, while achieving almost 2X higher system throughput compared to a multicore with all aggressive cores. This work appeared in the International Symposium on High-Performance Computer Architecture (HPCA) in February 2013 [14].

2.2.5. Core Assignment to Jobs

The current state of practice in supercomputer resource allocation places jobs from different users on disjoint nodes both in terms of time and space. While this approach largely guarantees that jobs from different users do not degrade one another's performance, it does so at high cost to system throughput and energy efficiency.

We proposed *Job Striping*, a technique that significantly increases performance over the current allocation mechanism by co-locating pairs of jobs from different users on a shared set of nodes. To evaluate the potential of job striping in large scale environments, the experiments are run at the scale of 128 nodes on the state-of-the-art Gordon supercomputer. Across all pairings of 1024 process NAS parallel benchmarks, job striping increases mean throughput by 26% and mean energy efficiency by 22%. On pairings of the applications GTC, LAMMPS, and MILC at equal scale, job striping improves average throughput by 12% and mean energy efficiency by 11%. In addition, we provide a simple set of heuristics for avoiding low performing application pairs. This work appeared in the Concurrency and Computation: Practice and Experience journal in 2013 [15].

2.3. Resiliency

2.3.1. Reliable Energy-Efficient On-Chip Networks

To build a power-efficient Thrifty chip, it is important to design a highly energy-efficient network. In an NTC environment, there are significant variations in speed and power consumption across the chip. Therefore, the network may have multiple voltage domains --- with high voltages for the slow regions and lower voltages for the faster regions. In addition, the speed of a given domain also changes dynamically, due to temperature changes caused by load variations.

To design the most energy-efficient network possible, we propose to dynamically change the voltage of each domain. In addition, to attain the minimum voltage that is tolerable at any time, we propose to monitor the timing error rate that occurs at each router and increase the voltage if the error rate is too high, or reduce the voltage if the error rate is zero or below a threshold. We are leveraging the fact that the network already has support to detect when an error occurs (e.g., a bit flip is detected by checksums, or a misrouted message is detected by a node whose ID does not match the one in the destination field). Moreover, the network has ways to tolerate the error (e.g., by retrying the message) without crashing. With this support, we can set the different voltages of the different domains at very-low values, adapting to the static and dynamic variation parameters at any time, while running the network fully reliably, and saving a lot of energy.

We have designed a micro-architecture composed of network sensors and actuators called *DynNet*. When a message is found to be erroneous at the destination node (e.g., the checksum does not match), DynNet increases the voltage of the routers in the path followed by the message by a small amount called Vdd_inc. This is because we do not know which router failed. As multiple messages following different routes go through a weak router and fail, that router's voltage will get incremented more than the others until it does not fail anymore. In addition, at regular (long) intervals, DynNet decreases the voltage of all the routers by a fixed amount called Vdd_dec. This is to ensure that the system does not continue to increase Vdd forever in an environment with temperature changes. Overall, with the right combination of Vdd_inc and Vdd_dec changes, the voltages of all the domains converge with time to the lowest (and therefore most energy efficient) values that still guarantee a reliable network.

We are evaluating a 16-router 2-dimensional torus global network for Thrifty. Each router has a three-stage pipeline. We have used our VARIUS-NTV models [2] to estimate the error rate of each router as a function of the voltage applied. For a high enough voltage, the error rate is 0; for a low enough voltage, the error rate is 1. Hence, as we decrease the voltage, the error rate follows a curve from 0 to 1. Figure 5 shows the resulting error rate for the second pipeline stage of each of the 16 routers. The second pipeline stage is the virtual channel allocator, and is the longest and most error-prone stage. There is one curve per router. We can see that process variation has a major impact on the different routers. Some routers can handle low voltage, while others not. The range of voltages is 0.45V to 0.65V.

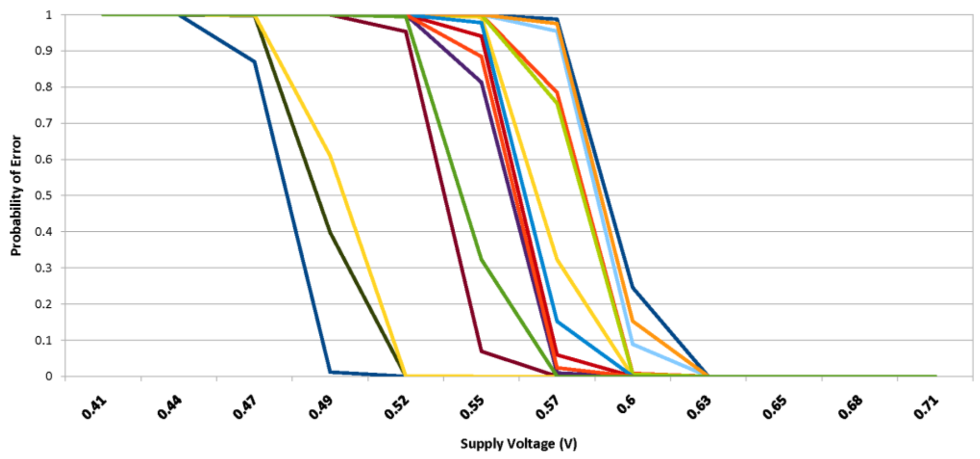


Figure 5. Error rates for different voltages applied to the 2nd pipeline stage of each of the 16 routers.

Figure 6 shows the voltages observed in the four routers in the diagonal of the torus, as a function of time. The X axis is the epoch number, where an epoch lasts 100ms. Initially, the voltage is about 0.8 V for the four routers. However, as the sensors sense the errors and the actuators change the voltage, the voltage of the four routers converges to different values: 0.64V, 0.63V, 0.61V and 0.57V. We can see that NetDyn is stable.

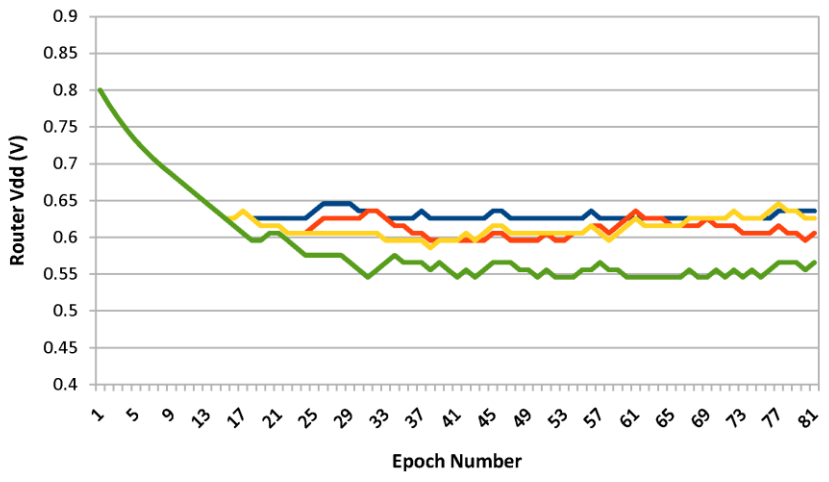


Figure 6: Changes in the voltage of 4 routers as a function of time.

Finally, Figure 7 shows the average power consumed by the 16-router network for different applications. For each application, the left bar is the baseline network (which includes NTC but without DynNet), while the right bar is DynNet. The last two bars show the mean. We can see that, without DynNet, the network consumes on average 0.225w, while with DynNet, it consumes 0.140w. This is a reduction of 40% of the network power, while still keeping the system completely reliable. This work is published under submission [5].

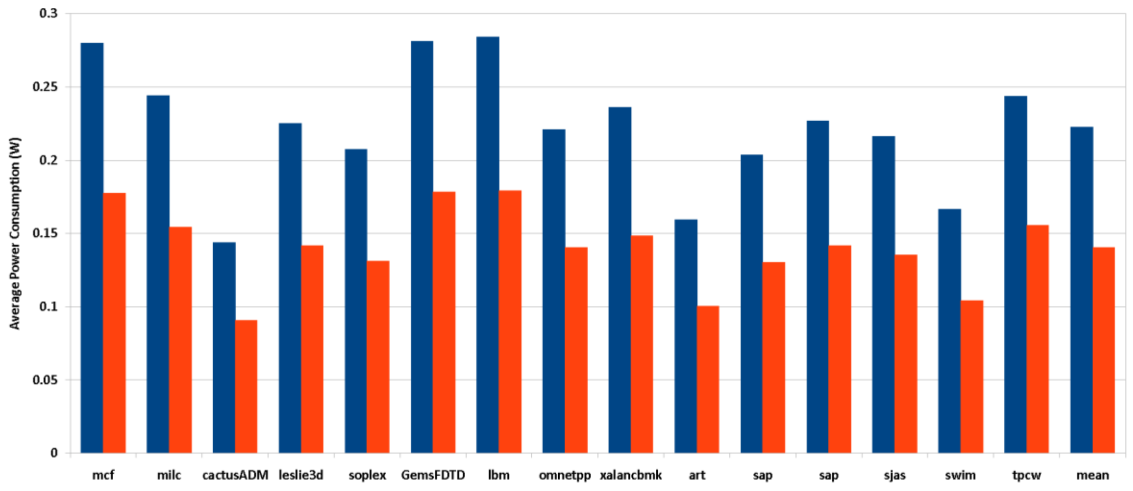


Figure 7: Average power consumed by the 16-router network without and with DynNet.

2.3.2. Compiler-Driven Transformations for Resiliency

We have developed a source-to-source translation framework for exascale fault-tolerance research. The framework is built using ROSE. It demonstrates the use of Triple Modular Redundancy (TMR) as an approach to provide HPC software with fault tolerance against transient faults, as we expect them to manifest themselves on future Exascale architectures. The results are published in [6]. The paper presents performance results showing that for a randomly selected subset of benchmarks the overhead of this extra layer of support is about 20%.

Figure 8 shows an example of a transformation of a loop into a TMR structure as it would be used in a simple approach. The transformation performs redundant computations and checks that all produce the same result. If they are different, a voting process occurs. It will be interesting to see if this approach may be competitive with future approaches to fault tolerance using check-point restart.

```
#pragma resiliency
for (int i = 1; i < arraySize-1; i++)
     a[i] = (a[i-1] + a[i+1]) / 2.0;
```

⇩

```
for (int i = 1; i < (arraySize - 1); i++) {
  int ii, correctCnt = 0;
  float aI[3] = {a[i], a[i], a[i]};
  #pragma omp parallel for
  for(ii = 0; ii < 3; ii += 1) {
     float aII[3] = {aI[ii], aI[ii], aI[ii]};
     // Original statement: aI[ii] =
     aII[0] = ((a[i - 1] + a[i + 1]) / 2.0);
     aII[1] = ((a[i - 1] + a[i + 1]) / 2.0);
     aII[2] = ((a[i - 1] + a[i + 1]) / 2.0);
     aI[ii] = aII[0];
     if (!(aII[2] == aII[1] && aII[1] == aII[0]))
        aI[ii] = (aII[0] + (aII[1] + aII[2])) / 3.00000F;
  }

  #pragma omp parallel for reduction (+:correctCnt)
  for(ii = (0); ii < 2; ii += 1)
      correctCnt += array_inter[ii] == array_inter[ii +
1];
  if (!(correctCnt == 2)) {
      printf("Result is not consistent across
executions...
      assert(false);
  }
}
```

Figure 8: Transforming a loop into a TMR structure.

3. Plans for Next 6-Months

Our main plan for the next six months is to further integrate the applications, compiler, and architecture/circuits layers efforts. Such integration is slow due to the fact that each of the layers works at a different granularity, and the need to focus on the novel aspects of the work (as opposed to reimplementing past work). We intend to speed up the integration along the following two lines, always using DOE applications to drive the evaluation:

1) Application and compiler work that drives the energy management API provided by the Thrifty architecture. The architecture provides novel ways of managing static and dynamic energy; the software should be able to drive it, possibly using compiler analysis or the knowledge of the idioms in the DOE applications.

2) Comparison of energy and performance between software managed caches and hardware coherent caches. We have modeled each of these structures in our simulators, and have performed work at individual layers (applications, compilers, architecture) comparing such structures. We will now integrate our efforts.

In addition to these main thrusts, we will continue to make individual contributions in each of the layers and publish novel work.

References:

[1] Ulya R. Karpuzcu, Abhishek Sinkar, Nam Sung Kim, and Josep Torrellas, “EnergySmart: Toward Energy-Efficient Manycores for Near-Threshold Computing”, International Symposium on High Performance Computer Architecture (HPCA), February 2013.

[2] Ulya R. Karpuzcu, Krishna B. Kolluru, Nam Sung Kim and Josep Torrellas, “VARIUS-NTV: A Microarchitectural Model to Capture the Increased Sensitivity of Manycores to Process Variations at Near-Threshold Voltages”, IEEE International Conference on Dependable Systems and Networks (DSN), June 2012.

[3] Aditya Agrawal, Prabhat Jain, Amin Ansari, and Josep Torrellas, "Refrint: Intelligent Refresh to Minimize Power in On-Chip Multiprocessor Cache Hierarchies”, International Symposium on High Performance Computer Architecture (HPCA), February 2013.

[4] Shah Mohammad Faizur Rahman, Jichi Guo, Akshatha Bhat, Carlos Garcia, Majedul Haque Sujon, Qingy Yi, Chunhua Liao, and Daniel J. Quinlan, "Studying The Impact Of Application-level Optimizations On The Power Consumption Of Multi-Core Architectures", ACM International Conference on Computing Frontiers 2012 (CF'12), May 15th-17th, 2012, Cagliari, Italy.

[5] Amin Ansari, Asit Mishra, Jianping Xu, and Josep Torrellas, “Designing Reliable, Ultra-Energy-Efficient On-Chip Networks Under Process Variation at Near-Threshold Voltage”, University of Illinois Technical Report, 2012.

[6] Jacob Lidman, Daniel J. Quinlan, Chunhua Liao, and Sally A. McKee, "ROSE::FTTransform - A Source-to-Source Translation Framework for Exascale Fault-Tolerance Research", Workshop on Fault-Tolerance for HPC at Extreme Scale (FTXS 2012), Boston, June 25-28, 2012.

[7] Benjamin Ahrens, Roger Golliver, and Josep Torrellas, “Exploring alternatives to hardware support for fine-grain synchronization”, in submission, 2013.

[8] C. Olschanowsky, M. Meswani, L. Carrington, and A. Snavely, ``PIR: A Static Idiom Recognizer'', *First International Workshop on Parallel Software Tools and Tool Infrastructures (PSTI 2010)* 2010.

[9] http://www.HYCOM.org/HYCOM/overview.

[10] http://physicsutahedu/~detar/milchtml.

[11] Nicholas P. Carter, Aditya Agrawal, Shekhar Borkar, Romain Cledat, Howard David, Dave Dunning, Joshua Fryman, Ivan Ganev, Roger A. Golliver, Rob Knauerhase, Richard Lethin, Benoit Meister, Asit K. Mishra, Wilfred R. Pinfold, Justin Teller, Josep Torrellas, Nicolas Vasilache, Ganesh Venkatesh, and Jianping Xu, “Runnemede: An Architecture for Ubiquitous High-Performance Computing”, International Symposium on High Performance Computer Architecture (HPCA), February 2013.

[12] Sanket Tavarageri, Wooil Kim, Josep Torrellas, and P Sadayappan, “Automatic Generation of Coherence Instructions for Software-Managed Multiprocessor Caches”, in submission.

[13] K. Seager, A. Tiwari, M. Laurenzano, J. Peraza, P. Cicotti, and L. Carrington, “Efficient HPC Data Motion via Scratchpad Memory”, International Workshop on Data-Intensive Scalable Computing Systems (DISCS-12), 2012.

[14] Amin Ansari, Shuguang Feng, Shantanu Gupta, Josep Torrellas, and Scott Mahlke, “Illusionist: Transforming Lightweight Cores into Aggressive Cores on Demand”, International Symposium on High Performance Computer Architecture (HPCA), February 2013.

[15] A. Breslow, L. Porter, A. Tiwari, M. Laurenzano, L. Carrington, D. Tullsen, and A. Snavely, “The Case for Colocation of HPC Workloads”, Concurrency and Computation: Practice and Experience, February 2013.

A. Appendix

We include some of the work that was reported in previous reports, that is still being developed or used in this period.

A.1. Compiler-Driven Code Transformations for Power Reduction

We have studied the impact of various application-level compiler optimizations on the power consumption of many applications running on an 8-core Intel workstation and a 32-core AMD workstation. We have used a wide range of sequential and multithreaded benchmarks. We have observed that application-level optimizations often have a much larger impact on performance than on power consumption. However, optimizing for performance does not necessarily lead to better power consumption, and vice versa. This work has helped us to gain the insight of impact of compiler optimizations on power consumption.

Figure 9 shows the Mflops (or MIPS) per watt of the average benchmarks. We can see that the actual optimization level used has a significant impact on the energy efficiency of the benchmarks. This work was published in [4].

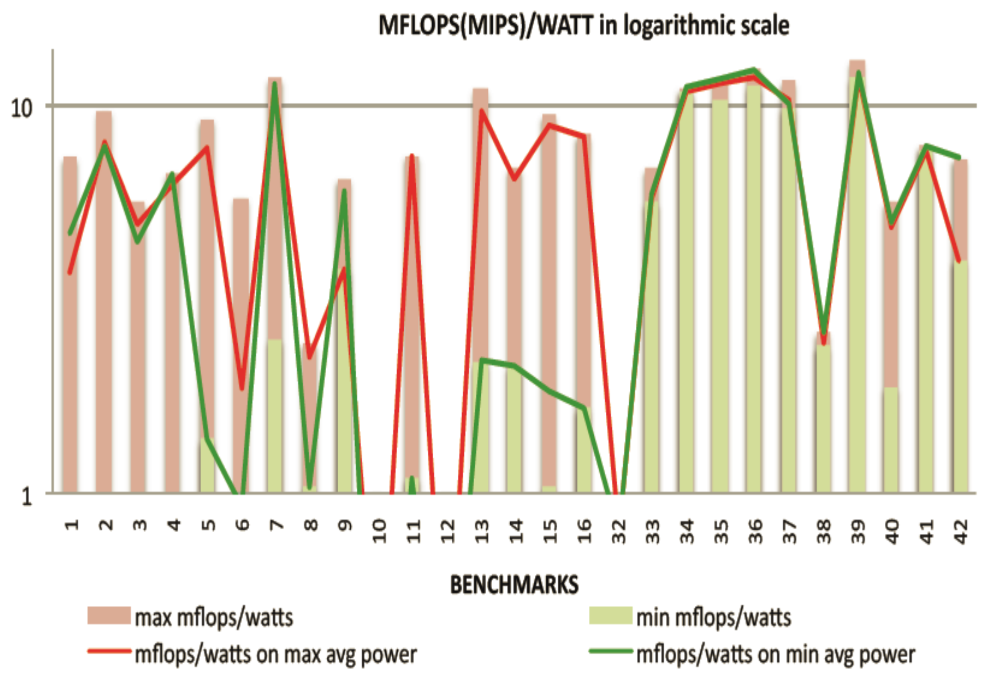


Figure 9: Impact of different compiler optimizations on the MFLOPS (or MIPS) per watt of codes.

A.2. NUMA-Aware Runtime System for Thrifty

We have developed a many-core NUMA-aware runtime system for exascale machines such as Thrifty. The library was released within the ROSE compiler in December 2011. The library supports data decomposition for multi-dimensional arrays, so that each portion of the arrays can be allocated to different memory banks in NUMA systems. The distributed storage of big arrays across different memory banks will improve the data locality on NUMA systems and, ultimately, performance.

A.3. Recognizing Idioms

PIR (PMaC Idiom Recognizer)[8] is a tool for searching source code for idioms. An idiom is a local pattern of computation that a user may expect to occur frequently in certain applications. For example, a stream idiom is a pattern where memory is read from an array, some computation may be done on this data, and then the data is written to another array. A stream reads sequentially from the source array and writes sequentially to the destination array. A stream may arise from the presence of the statement A[i] = B[i] within a loop over i.

Idioms are useful for describing patterns of computation that have the potential to be optimized, for example, by loading the piece of code to a coprocessor or accelerator.

The PIR tool allows us to automate searching for idioms in a powerful way by using data-flow analysis to augment the identification process. It would be very difficult to use a simpler searching tool, such as regular expressions, because a regular expression does not naturally discern the meaning of the text it identifies. For example, in the code shown in **Error! Reference source not found.**0, a simple regular expression based on (for example) "grep" that searches for stream idioms of the form "A[i]= B[i]" would incorrectly identify line 1 as a stream and it would miss the stream at lines 3-4 because the assignment is broken into multiple statements.

```
1. values[c] = constants[c];
2. for( i = 0; i < 10; ++i ) {
3.   item = source_array[i];
4.   dest_array[i] = item;
5. }
```

Figure 10: Sample stream idiom code.

PIR, however, is able to determine that line 1 is not in a loop and that c is a constant. This indicates that the meaning of this statement is simply a variable assignment, rather than a stream. In lines 3-4, PIR uses data-flow analysis to determine that item in line 4 holds a value from the source array making this a stream.

PIR's design provides the flexibility to identify optimization opportunities for many different hardware configurations. The user provides descriptions of the idioms to be identified. As a starting point, PIR provides a set of commonly useful idioms and access to an Idiom definition syntax that allows for user customization of the idioms.

PIR includes seven idiom definitions we have found to be common in HPC applications. The user is free to define more via a simple pattern describing API. The pre-defined idioms are described in the following. All of the code samples are assumed to be part of a loop, i (and j) are loop induction variables.

- Stream: A[i] = A[i] + B[i]

The stream idiom includes accesses that step through arrays. In the above example two arrays are being stepped through simultaneously, but the stream idiom is not limited to this case. Stepping through any array in a loop where the index is determined by a loop induction variable is considered a stream.

- Transpose: A[i][j] = B[j][i]

The transpose idiom involves a matrix transpose, essentially reordering an array using the loop induction variable.

- Gather: A[i] = B[C[i]]

The gather idiom includes gathering data from a potentially random access area in memory to a sequential array. In this example the random accesses are created using an index array, C.

- Scatter: A[B[i]] = C[i]

The scatter idiom is essentially the opposite of gather. Values are read from a sequential area of memory and saved to an area accessed in a potentially random manner.

- Reduction: s = s + A[i]

A reduction can be formed from a stream, as in the working example, or a gather. It implies that the value returned from the read portion of the idiom is assigned to a temporary variable.

- Stencil: A[i] = A[i-1] + A[i+1]

A stencil idiom involves accessing an array in a sequential manner, including a dependency between iterations of the loop.

File Name	Line #	Function	Idiom	Code
mod_tides.F	623	tides_set	gather	pf(i)=f(index(i))
mxkrt.f	992	mxkrtbaj	reduction	sdp=sdp+ssal(k)*q

Table **Error! Reference source not found.**: Sample output from PIR analysis on HYCOM.

Table **Error! Reference source not found.** presents just a sample of the report for an application. The sample shows how PIR is able to classify the idiom, capture the source file, function name and even the line number of source code used for the identification (additional information about loop depth, start, and end are captured but not shown).

The PIR user manual and programmers guide can be found online at www.sdsc.edu/pmac. **Error! Reference source not found.** shows the PIR results when combined with runtime to calculate the runtime contribution of each idiom. Two applications were used for this: HYCOM and Milc. HYCOM [9] is a popular ocean modeling code. Milc [10] is a code for simulations of SU3 lattice gauge theory on MIMD parallel machines.

Idiom	**HYCOM (8cores)**	**Milc (8cores)**
Gather/scatter	14.2%	1.2%
Stream	21.1%	5.6%
Reduction	0.0%	15.7%
Stencil	4.7%	0.0%
Transpose	0.9%	0.0%
Mat-Mat Mult	23.7%	61.2%
Mat-Vec Mult	0.0%	10.5%
Fraction of loops Covered	**64.6%**	**94.2%**

Table 2: Idiom runtime contribution.

A.4. Evaluating Idioms on a Scratch Pad

Another memory hierarchy organization that can be used to execute some codes faster and more energy-efficiently is a scratch pad. A scratch pad is an on-chip memory module where the compiler (or other software) has complete control of where to place the data. Since our PIR tool detects the idioms in the code, we can then map them efficiently on scratch pads. The key is to develop an accurate estimate of which idioms will perform well on the scratch pad memory. In this work, we develop a general methodology to characterize idiom operations on scratch pad memory.

Building a model of the interaction of the hardware and the application requires two main components: the model/machine component and the application component. The model/machine component involves measuring or estimating the performance and energy benefits of using the scratch pad memory for the different idioms and identifying the parameters that affect performance (i.e. locality, access pattern, data movement, etc.). The application component entails automating the detection of idiom operations in a large scale HPC application and simulating these instances on the scratch pad memory to capture instance specific information for the models. The final step combines the machine/model and application components to provide the details of performance and energy changes of the application on the Thrifty architecture with the scratch pad memory.

Early Proof-of-Concept Study

In order to have a general scheme for modeling and predicting idiom operation performance, we first need to identify instances of the idioms within the source code. Secondly, we need to simulate each instance in the application on the scratch pad memory to collect the relevant model data.

The PIR tool, described above, was used to automate the search for the idiom instances in large scale scientific application. Once the idiom instances are detected the second step is simulating each instance on the scratch pad memory. In order to be able to simulate these applications running at full-scale a scratch pad memory simulator was built on-top of the PEBIL (PMaC's Efficient Binary Instrumentor for Linux) toolkit. PEBIL is an open source binary instrumentation toolkit for x86/Linux. Its basic block analysis tool provides static and dynamic information about each basic block (e.g., visit counts, loop and function membership information). It is designed to instrument an identified set of basic-blocks in an application and capture the memory addresses of those blocks during the execution of the application. To conserve time and space the address stream is simulated on-the-fly while the application is running. The tools developed at PMaC work on an application's binary. This allows us to simulate an application's execution on a variety of different memory configurations without having to touch the application source code. Thus, allowing for scratch pad experiments on large scale applications.

In order to perform this simulation for the idiom operations identified by PIR, the following two steps are required. First the information gathered from PIR needs to be translated and conveyed between the two tools because PIR works on source code while PEBIL works on the binary—we need to identify the binary code corresponding to the source code. This translation allows PEBIL to add additional instrumentation to those basic-blocks identified by PIR. Secondly, only those basic-blocks identified should be simulated on the scratch pad memory simulator.

In order to translate PIR identified idioms to corresponding basic-blocks in the application binary, a basic block analysis tool written on top of the PEBIL toolkit is utilized. This tool provides a mapping between the source file and line number and the basic-blocks. Information collected by PIR maps idioms to source line numbers. Our current approach to mapping the PIR identified idioms to the basic blocks in application's binary is manual and uses simple scripts that combine the information provided by the two tools. Once the basic blocks are identified, the scratchpad simulation tool simulates the execution of the idioms.

Preliminary Results

In order to investigate those idioms which might benefit from scratchpad memory, a benchmark suite was developed. The current suite contains 3 idioms: gather/scatter, stream, and reduction. In the coming year the suite will be expanded to include: matrix-matrix multiplication, matrix-vector multiplication, stencil, transpose, and others. By using the idiom benchmark suite we can explore how those idioms perform on the scratch pad simulator, to identify the parameters and idioms which will benefit from the scratch pad memory.

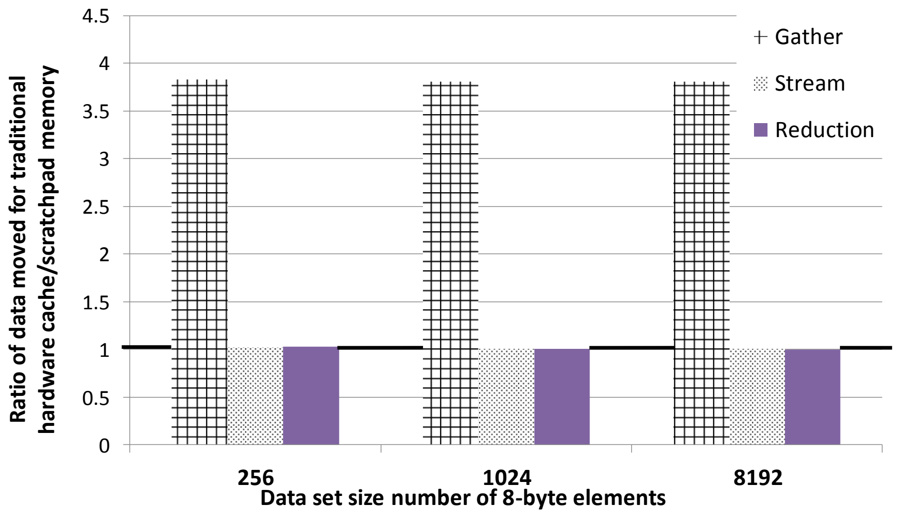


Figure 11: Data movement vs. data set size of idioms on scratchpad and traditional cache

Figure 11**: Data movement vs. data set size of idioms on scratchpad and traditional cache**

plots the data movement for three key idioms: stream, gather/scatter, and reduction. The y-axis represents the ratio of the data moved for a traditional cache to that of the scratch pad memory. Thus, ratios greater than one indicates those idioms which benefit from scratch pad and the level of such benefit. The x-axis represents the size of the memory footprint for the idiom. It is clear from the figure that one idiom shows a clear benefit from the scratch pad is the gather/scatter (G/S) idiom.

Next we chose the G/S idiom to do a preliminary test of the tools on a full-scale application. The initial focus is on the G/S idiom due to its ability to exacerbate a systems memory performance. In a Gather, non-contiguous memory locations are collected up into a contiguous array; in a Scatter, contiguous array elements are distributed to non-contiguous memory locations; because these types of operations are 1) prevalent in many scientific applications 2) performance-limited on many architecture by the latency of main memory, various architectural features have been proposed to try to accelerate them.

Using the PIR tool and the HPC application HYCOM we identified 33 instances of gather/scatter in the source code for HYCOM. Matching the static source code information from PIR to the basic-block information from PEBIL we then instrumented 33 basic-blocks of HYCOM to be run through the scratch pad simulator. The results of this simulation are show in **Error! Reference source not found.** below.

APPLICATION	DATA moved by cache	DATA moved by scratch pad
HYCOM	93.72 MB	74.45 MB

Table 1. Simulated data movement of traditional cache and scratch pad memory.

The results of this preliminary study show that by using a scratch pad memory for all instances of gather/scatter within HYCOM we could reduce the amount of data movement by 25%. This is just a preliminary test of the tools to understand the types of investigations that are possible.

A.5. A Model for Increased Process Variation Sensitivity at NTC

NTC is more sensitive to process variations than the conventional Super-Threshold Computing (STC). The result is higher power consumption and lower frequencies than would otherwise be possible, and potentially a non-negligible fault rate. To help address variations at NTC at the architecture level, we have developed the first micro-architectural model of process variations for NTC. The model, called VARIUS-NTV, models how variation affects the frequency attained and power consumed by cores and memories in an NTC manycore, and the timing, hold, and stability faults in SRAM cells at NTC. The key aspects include: (i) adopting a gate-delay model and an SRAM cell type (8-transistors) that are tailored to NTC, (ii) modeling SRAM failure modes emerging at NTC, and (iii) accounting for the impact of leakage current in SRAM timing and stability models. In particular, VARIUS-NTV models all of the SRAM failure modes (except read upsets, which cannot occur in the 8-transistor cell because a read cannot flip the cell contents by construction). They include hold failure, write stability failure, read timing failure, and write timing failure.

We have used the model to estimate variations in many future chips, and compared the variations at NTC and STC. We have also validated the model against published variation measurements of the Intel's 80-Core TeraFLOPS processor. As an example, we show here the results of evaluating a simulated 11nm, 288-core tiled many-core (Figure 12) at both NTC and STC. The many-core is organized in 36 tiles for ease of design. Each tile has a tile memory and 8 cores, each with a per-core memory. Each core is a single-issue engine where memory accesses can be overlapped with each other and with computation. The cores are connected with a bus inside each tile and with a 2D torus across tiles.

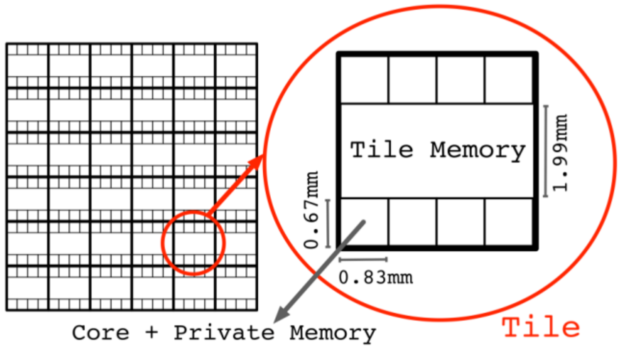
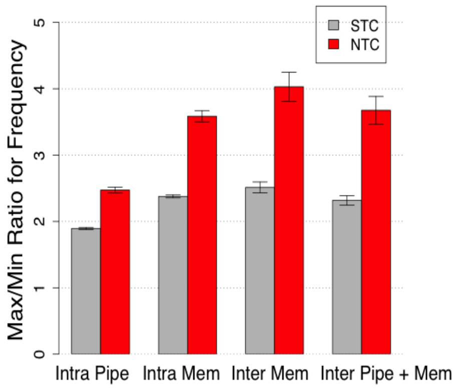


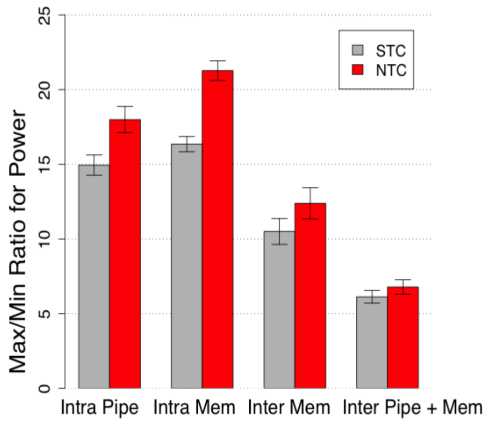
Figure 12: Example of a Thrifty chip.

We consider three types of on-chip components separately: logic (the core pipelines), small memories (the per-core local memories) and large memories (the per-tile memories). We do this because they have different critical path distributions. We consider intra-tile variations first. In each tile, we compute the ratio of the frequencies of the fastest and slowest pipelines in the tile. We then take the average of the ratios across all tiles (Intra Pipe). We repeat the same process for local memories in the tile to calculate Intra Mem. Finally, we compute the power as the sum of their static and dynamic power. We take the power ratio of highest to lowest consuming pipelines, and highest to lowest consuming local memories, to compute Intra Pipe and Intra Mem, respectively.

For inter-tile variations, we measure the ratio of the frequencies of the fastest and slowest tile memories on chip (Inter Mem). We then consider the frequency that each tile can support (the lowest frequency of its pipelines, local memories and tile memory), and compute the ratio of the frequencies of the fastest and slowest tiles (Inter Pipe+Mem). Finally, we repeat the computations for power (Inter Mem and Inter Pipe+Mem). We report the mean of the experiments for 100 chips. The results are shown in Figure 13(a) and 13(b).



(a)



(b)

Figure 13: Comparing the variation in frequency (a) and power (b) in STC and NTC.

Figure 13(a) compares these frequency ratios for NTC and STC. We observe that the frequency ratio of the fastest to the slowest components is substantially higher at NTC than at STC — for the same process variation profile. For example, Inter Pipe+Mem at NTC is 3.7, while it is only 2.3 at STC. This is because a low Vdd amplifies the effect of process variations on delay. Figure 13(b) shows the power ratios. The variation in total power also increases at NTC. However, the relative difference in power ratios between NTC and STC is generally smaller than the relative difference in frequency ratios. The reason is that power includes both dynamic and static power, and the ratios for static power are the same for NTC and STC. Consequently, the relative difference in power ratios is smaller. Still, the absolute difference is significant. Consequently, the chip is more heterogeneous at NTC. This work appears in [2].