Runnemede: An Architecture for Ubiquitous High-Performance Computing

Nicholas P. Carter[1,4], Aditya Agrawal[1,2], Shekhar Borkar[1], Romain Cledat[1], Howard David[1], Dave Dunning[1], Joshua Fryman[1], Ivan Ganev[1], Roger A. Golliver[1], Rob Knauerhase[1], Richard Lethin[3], Benoit Meister[3], Asit K. Mishra[1], Wilfred R. Pinfold[1], Justin Teller[1], Josep Torrellas[2], Nicolas Vasilache[3], Ganesh Venkatesh[1], and Jianping Xu[1]

[1]Intel Labs, Hillsboro, Oregon
[2]University of Illinois at Urbana-Champaign, Champaign, Illinois
[3]Reservoir Labs, New York, New York
[4]Contact email: nicholas.p.carter@intel.com

Abstract

DARPA's Ubiquitous High-Performance Computing (UHPC) program asked researchers to develop computing systems capable of achieving energy efficiencies of 50 GOPS/Watt, assuming 2018-era fabrication technologies. This paper describes Runnemede, the research architecture developed by the Intel-led UHPC team. Runnemede is being developed through a co-design process that considers the hardware, the runtime/OS, and applications simultaneously. Near-threshold voltage operation, fine-grained power and clock management, and separate execution units for runtime and application code are used to reduce energy consumption. Memory energy is minimized through application-managed on-chip memory and direct physical addressing. A hierarchical on-chip network reduces communication energy, and a codelet-based execution model supports extreme parallelism and fine-grained tasks.

We present an initial evaluation of Runnemede that shows the design process for our on-chip network, demonstrates 2-4x improvements in memory energy from explicit control of on-chip memory, and illustrates the impact of hardware-software co-design on the energy consumption of a synthetic aperture radar algorithm on our architecture.

1. Introduction

DARPA's Ubiquitous High-Performance Computing (UHPC) program challenged researchers to develop hardware and software techniques for *Extreme-Scale* systems: computing systems that deliver 100–1,000x higher performance than current systems of the same physical footprint and power consumption [34]. Extreme-scale systems should deliver the energy-efficiency (50 GOPS/Watt), reliability, and scalability required to construct practical exaOP supercomputers (machines that execute 10^{18} operations/second) in the 2018-2020 timeframe.

In this paper, we describe *Runnemede*, the research architecture developed by the Intel-led UHPC team. Runnemede's goal is to explore the upper limits of energy efficiency *without* the constraints imposed by backward compatibility and the need to support conventional programming models. Its hardware, OS/runtime, and applications are being developed through a co-design process to produce a system in which hardware and software work together to maximize performance and minimize energy consumption.

We begin this paper by outlining our technical approach. We then present Runnemede's architecture, focusing on the hardware but describing the software stack where relevant. This is followed by a preliminary evaluation of our network design, memory system, and the impact of our co-design process. Finally, we present related work and conclude.

2. Technical Approach

Runnemede[1] is heavily influenced by several predictions about 2018-2020 fabrication technology. The power consumed by logic is expected to scale well as feature sizes

[1]Following Intel tradition of naming projects after geographic locations in the US or Canada, the Runnemede project was named after Runnemede, NJ, inspired by Runneymede, England, where the Magna Carta was signed.

shrink, but not as well as transistor density, leading to the design of *overprovisioned*, *energy-limited* systems that contain more hardware than they can operate simultaneously. Signaling power is expected to scale much less well than logic power, making on-chip and off-chip communication a larger fraction of overall power. SRAM power is also expected to scale less well than logic power, due to the difficulty of designing SRAM circuits that can operate at low supply voltages. We expect the power consumed by DDR DRAMs to decrease relatively slowly over time, although stacked DRAMs with improved interfaces will become available in the not-too-distant future [28], significantly reducing DRAM power consumption. However, technology will limit the number of DRAM die per stack, and the need to provide I/O pins for each stack will limit the total DRAM capacity of systems that use stacked DRAMs, potentially leading to systems that combine stacked and DDR DRAMs into two-level DRAM hierarchies.

These technology trends lead to several predictions about extreme-scale computer systems. Extreme-scale computer systems will be energy-limited and overprovisioned. Therefore, to achieve maximum performance, they should be designed such that their key subsystems (cores, memory, and networks) can each consume a disproportionate share of the system's full power budget when that subsystem is the limiting factor on performance. This, in turn, requires that software and hardware actively manage their power consumption to ensure that the system stays below budget. The computer industry is already seeing early examples of overprovisioned designs, such as Intel's Turbo Boost [17] technology, which defines a base clock rate for each chip that meets the chip's power budget when all cores are active, and increases the clock rate when some cores are idle to provide high performance on both serial and parallel codes/regions.

Because extreme-scale systems will be energy-limited, they must be designed to operate at their most-efficient supply voltages and clock rates (low, near-threshold voltage (NTV) and modest frequency), although the ability to increase supply voltage and clock rate on serial sections of code will also be of great benefit. Operating at low clock rates implies that extreme-scale systems will require more parallelism than current systems to deliver a given amount of performance, leading to interest in execution models that maximize parallelism and minimize synchronization.

Energy-limited systems are expected to be heterogeneous, because there is no performance benefit to building more of a given unit, core, or module than can be operated simultaneously (possibly plus a few spares for reliability), which is sometimes described as the "dark silicon problem/opportunity" [6]. Being energy-limited also encourages specialization, or the design of hardware that may be used infrequently, as long as it can be powered off when not in use. A small number of custom functions/instructions can significantly improve an architecture's efficiency and performance on applications that make use of those functions [15] [35], and including such functions does not decrease the performance of applications that do not use them if the system is limited by energy instead of area.

Finally, the technology scaling trends described above imply that energy-limited systems must minimize data movement in order to achieve maximum performance. While stacked DRAM is expected to require much less energy to access than DDR DRAM, on-chip wires and SRAMs are expected to scale less well than logic, making data movement increasingly expensive relative to computation. This argues that extreme-scale memory systems and applications should focus on bandwidth efficiency by eliminating unnecessary data transfers.

2.1. Energy-Efficiency from the Ground Up

The Runnemede architecture 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 processor 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 [19], and on designing circuits and organizations to tolerate them in an energy-efficient manner. In addition, Runnemede 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, Runnemede's processor chip includes a large number of relatively-simple cores. The initial design described in Section 4 takes this philosophy to the extreme of single-issue, in-order cores, although future work will explore the trade-offs involved in different core architectures. 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.

Runnemede'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 Runnemede 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.

The software system is co-designed with the hardware, and provides a number of programming models with different trade-offs between simplicity and control of the underlying hardware. Our higher-level models include Hierarchically-Tiled Arrays (HTAs) [13], which expresses computations as blocks or tiles in successive, hierarchical levels, and Concurrent Collections (CnC) [7], which describes computations in a high-level dataflow-like manner. The R-Stream® compiler can automatically generate parallelized and locally-optimized code for sequential loop nests. Finally, for programmers who want a lower-level interface to the hardware, it is possible to code directly to our runtime's codelet model, which is described in the next section.

Runnemede also includes mechanisms for energy-efficient resilience. In particular, we envision an incremental in-memory checkpointing system [2], which can be adapted to take advantage of the structure of CnC programs

2.2. Hardware-Software Co-Design

Runnemede is a co-designed hardware/software effort, in which the hardware, execution model, OS/runtime, and applications are being developed simultaneously by a team that combines computer architects, system software experts, compiler developers, and application experts. This approach is made easier by the fact that the UHPC program defines five "challenge problems" that represent a significant fraction of the anticipated extreme-scale applications.

Given the extreme amounts of parallelism required to achieve exaOP performance at near-threshold supply voltages, we are designing Runnemede's runtime system around a dataflow-inspired [30] execution model. In contrast to pthread-style execution models, which implement parallel programs using long-running communicating threads, dataflow-inspired execution models represent programs as graphs of (typically short-running) tasks, where edges represent dependencies between tasks. This is similar to the way dataflow processors represent programs as graphs of instructions, with edges that represent dependencies. Further, in our execution model, the tasks in a program graph are *codelets* [14] — self-contained units of computation with clearly-defined inputs and outputs. Codelets are assumed to run to completion once they begin executing, although the operating system may intervene to halt a codelet that has entered an infinite loop or otherwise exceeded the "acceptable" codelet execution time.

Dataflow-inspired execution models have a number of characteristics that make them well-suited to extreme-scale systems. First, they make it easy for each phase of an application to exploit all of the parallelism available to it instead of encouraging a static division of an application into threads. Second, in dataflow execution models, only the producer and the consumer(s) of an item need to synchronize, potentially reducing synchronization costs. Third, the non-blocking "complete or fail" nature of codelets allows us to avoid much of the context-switching overhead of traditional OSes. Finally, a dataflow-inspired execution model makes it easy to identify a computation's inputs and outputs, and thus to schedule code close to its data, to marshal input data at the core that will perform a computation, and to distribute results from producers to consumers. This well-specified data movement both motivates and supports our decision to use software-managed on-chip memories.

Just as hardware issues affect our choice of execution model, software concerns influence hardware design. Having an execution model in which tasks (codelets) have well-defined inputs encourages the design of cores with separate power and clock gating for memories and execution units. This allows the runtime to turn on a core's memory(ies) in order to marshal a codelet's inputs and only turn on the execution units when input data is available and the codelet is ready to execute, thus avoiding the energy that would be wasted by idle execution units waiting for inputs to arrive.

Our execution model also influences our hardware design by encouraging the design of two types of cores: general-purpose Control Engines (CEs), which execute OS/runtime code, and energy-optimized Execution Engines (XEs), which execute codelets from user applications. With a space-separated (rather than time-separated) division between system code and user code [24], hardware to enforce protection rings (e.g., user/kernel mode) can be safely omitted from the CEs. Additionally, the non-blocking property of codelets means that I/O operations can only occur on inter-codelet boundaries, allowing a model in which the XEs do not contain I/O hardware. Instead, an I/O operation is represented as a dataflow dependence between two codelets. When the producer codelet reaches the I/O operation, it terminates with a request that a CE perform the I/O operation. When the I/O operation completes, the runtime notes that the consumer codelet's data dependence has been satisfied, and schedules the consumer for execution.

Co-design also affects our resilience and power-efficiency schemes. Our hardware provides the runtime

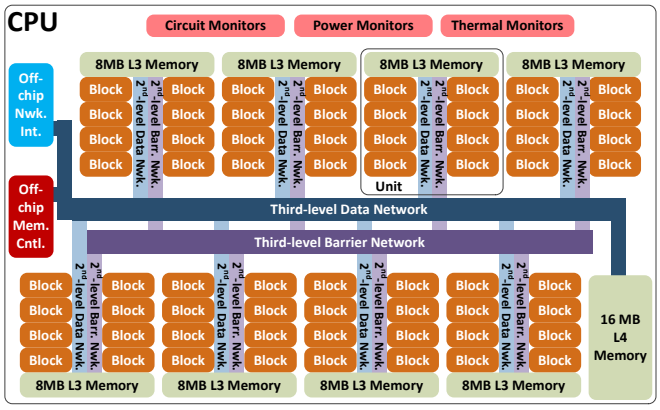

Figure 1. Runnemede chip architecture.

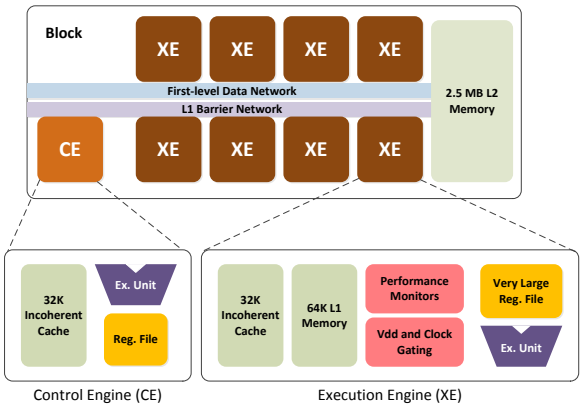

Figure 2. Contents of a block.

with information about the status, temperature and power usage of different regions on the chip. This allows the runtime to allocate work to avoid overheating, reducing hardware complexity. Similarly, the runtime is able to react to hardware failures, for example by assigning less-parallel tasks to a region that contains a failed core, or by increasing the region's clock rate to compensate if latency is important.

Finally, having a well-designed set of target applications allows us to co-design the hardware and the applications. We implement a number of instructions, such as the `sincos` instruction described later, that have a significant impact on one or more of the challenge applications. Application characteristics also influence our network design, including its barrier hardware and support for collective operations, and our synchronization primitives. In turn, understanding the hardware design allows our application experts to tune their algorithms to the strengths of the hardware.

3. Runnemede Architecture

As illustrated in Figure 1, the Runnemede architecture is modular and hierarchical, which allows applications to take advantage of locality and makes it easy to scale the architecture to a wide range of performance, price, and chip-size points. The basic module of Runnemede is the *block* (shown in Figure 2), which contains several cores, the first-level networks, and an L2 scratchpad memory. The next level of the hierarchy is the *unit*, which contains multiple blocks, an L3 scratchpad, and the second-level networks. A full Runnemede chip would contain multiple units and an L4 scratchpad, which would be connected by the third-level networks, allowing hundreds of cores to be integrated onto a chip. An off-chip network port allows multiple Runnemede chips to be integrated into a single-board system, with larger systems consisting of multiple boards.

3.1. Architecture of a Block

Each block is a heterogeneous system that contains one Control Engine (CE), which executes operating system and runtime routines, and multiple Execution Engines (XEs), which execute tasks from application programs. The 8 XEs shown in the figure are an initial estimate of the number of XEs that one CE will be able to support without the CE becoming the performance bottleneck; the number of XEs/block in an actual design will depend on the amount of CE support each codelet requires. Similarly, the 8 blocks per unit shown in Figure 1 is an early estimate that will be revised as we gather data about how much L3 memory is required to support a block of cores and the amount of locality in extreme-scale applications.

This heterogeneity is expected to increases energy-efficiency by allowing us to optimize each type of core for the work it does. XEs can be optimized for performance/watt on parallel computations, while CEs are optimized for more latency-sensitive OS operations. Infrequently-used hardware, such as I/O, can be placed in the CE to improve XE efficiency on computation kernels. Finally, separating the XE and the CE simplifies the design of systems in which the XEs in different blocks are optimized for different types of computations, since blocks with different types of XE would present the same interface to the OS and hardware outside the block.

CEs in Runnemede are typically general-purpose processor cores. XEs are typically custom architectures, containing one or more execution pipelines, a large (512-1024 entry) register file, a software-managed L1 scratchpad and an *incoherent* cache. XE instructions may be stored either in the cache or the scratchpad, as selected by a mode bit. Each of the scratchpads and register files in a Runnemede system maps onto a unique range of addresses from a single shared address space that is described in more detail in Section 3.2.

Each XE also contains performance monitors and registers that control power and clock gating, which are also mapped into the address space.

To reduce XE-CE communication overhead, each XE contains a set of memory-mapped registers for fast XE-CE communication. Writes into these registers inform the CE that the XE needs attention and pass information about what the XE needs the CE to do. If the CE requires additional information to handle the XE's request, it can read that information directly from the XE's memory or registers.

3.2. Memory Hierarchy and Address Space

Runnemede's on-chip memory hierarchy does not have hardware-coherent caches. While coherent caches simplify programming, their fixed line lengths and replacement policies can make them energy-inefficient if an application's access patterns do not match the cache's assumptions.

Instead, most of Runnemede's on-chip memories are software-managed scratchpads (blocks of SRAM that are mapped onto distinct regions of the address space so that software, rather than hardware, selects which data is kept in each scratchpad). This approach can significantly increase the energy-efficiency of some codes by eliminating transfers of unused data, false sharing, set conflicts, and collisions between streaming data and high-locality data — albeit at a potentially non-trivial cost in programming effort. To simplify the use of the scratchpads, Runnemede provides a set of DMA-like block transfer operations. These operations also reduce energy by performing cache-line-wide accesses to DRAM, which are much more energy-efficient than single-word accesses.

Runnemede also includes incoherent, software-managed, caches in each core, which are accessed via `load.cache` and `store.cache` instructions. The software-managed caches are intended to provide an intermediate efficiency/programming effort point between scratchpads and hardware-coherent caches. In our software-managed caches, hardware manages fetching and writeback of lines from/to whichever scratchpad, register file, or DRAM the address being referenced maps onto, but software is responsible for maintaining consistency when multiple caches may contain copies of the same location. To assist in this, Runnemede provides several cache-management instructions that prefetch lines, invalidate them (remove them from the cache without writing back dirty data), update them from the backing store, or evict them without writing back dirty data.

Runnemede provides a single, 64-bit, address space that is shared by all of the software running on the machine. Each scratchpad, DRAM, and register file in the system, as well as all of the performance monitors and control registers, appears in the address map. Runnemede implements a physical address space, with no virtual memory. This eliminates both the energy costs of address translation and the limits that TLB capacity places on the amount of memory a program can access efficiently (without requiring additional memory accesses for page-table walks).

Using a physical address space eliminates the energy cost of translation, but also eliminates the protection and relocation benefits of virtual memory. In addition, a physical address space has the potential to require the use of 64-bit addresses everywhere in the memory system. We solve the latter problem with hardware that determines the number of address bits required for each request based on the distance to the memory being referenced and only sends that many bits over the network, greatly reducing the number of address bits transmitted in programs with high locality.

To provide protection without translation, each block in a Runnemede architecture incorporates a "gate" unit that can be configured to allow or deny requests by a given set of cores to given ranges of the address space. Moreover, our runtime provides relocation by allocating memory as "data blocks", each of which has a unique identifier. Before a codelet's first use of a data block, it must call a translation routine that returns the physical address of the start of the data block. This allows the runtime to relocate data blocks as long as no codelet is currently referencing them, and limits translation costs to one translation per codelet per data block, as opposed to one translation per memory reference. Finally, our future research will investigate address translation mechanisms that impose less overhead than traditional virtual memory [12][33], in order to make it easier to map large data structures across multiple physical memories.

3.3. Networks

A Runnemede processor contains two independent hierarchical networks: a data network and a barrier/reduction network. Each block contains its first-level data and barrier networks, which provide low-energy communication within the block. Each block's first-level networks also interface with the second-level networks that connect the blocks in a unit, which in turn interface with the third-level networks that connect the units on a chip. This hierarchical network design allows Runnemede to provide *tapered* bandwidth, such that the amount of bandwidth between two points is inversely proportional to the distance between them and thus to the energy per bit of messages.

The data network handles the traffic generated by ordinary memory references. When a core references data from a memory located outside of the core, the hardware creates a request message that is transmitted to the destination memory over the data network, instructing it to perform the memory access and return the result. In contrast, the barrier/reduction network provides a mechanism for fast bar-

riers and reductions (operations that combine inputs from many cores into one output), and can also be used to perform broadcast/multicast operations. Performing a barrier or a reduction using this network is a two-phase process that separates arriving at a barrier or at the start of a reduction from waiting for the barrier/reduction to complete. This allows a task to signal other tasks that they can proceed past the barrier/reduction at the earliest possible point, perform any independent work that it may have, and then only wait for other tasks to reach the barrier/reduction when it becomes absolutely necessary, reducing wait times.

3.4. Power Management

Since an overprovisioned system incorporates more hardware than it can simultaneously operate, it must also contain mechanisms that allow it to dynamically allocate power to different portions of the system in order to remain within its power budget. Runnemede's power management mechanisms provide fine-grained control over clock rates, supply voltages, and power/clock gating. They are integrated into our address map to make them easy to access.

3.4.1. Dynamic Voltage/Frequency Scaling

The Runnemede architecture divides each CPU into several independently controllable clock/power domains, each containing one or more blocks, depending on the size of the chip and the number of voltage controllers it is feasible to fabricate, with an additional domain for the on-chip network. This provides fine-grained control over voltage and frequency, and also allows the network to operate at a voltage/frequency point that minimizes errors while the cores operate at the most energy-efficient voltage/frequency.

3.4.2. Power and Clock Gating

Runnemede provides clock-gating (clock is disabled but power supply is on, so that the unit retains its state) and power-gating (power supply and clock are disabled, which destroys the state of the unit) at multiple levels of granularity. Individual cores, memory modules, and network components can be clock-gated or power-gated independently. In addition, portions of a core may be power- or clock-gated independently, allowing software to disable units that it knows it will not use, such as the floating-point unit during an integer computation. This also allows software to power on a core's memory and ALUs at different times, to minimize power consumption while waiting for input data.

3.4.3. Power Management Interface

Runnemede's power management mechanisms are controlled through a set of memory-mapped registers that appear in the global address space. Reading these registers returns information about the state of the appropriate unit, while writing them changes that state. To prevent malware or buggy code from selecting power states that exceed the chip's thermal budget or interfere with other applications, we use a combination of our gate-based memory protection mechanisms and hardware-enforced limitations on which registers user code may write to.

Looking forward, one of the challenges in our future work will be developing runtime systems and hardware-software interfaces that use our power management interface to configure the chip to deliver the best performance per Watt for each application. For example, applications that achieve good parallel speedup will maximize performance per Watt by running on many cores but at low clock rates and supply voltages, while applications with poor parallel speedup will prefer a small number of cores at high clock rates. To achieve optimal efficiency, extreme-scale systems will need mechanisms to determine which category each application or phase of an application falls into, how to allocate their power budget across the applications running on a system, and how to configure the resources available to each application to best make use of its power budget.

3.5. Resilience and Reliability

Resilience is a major challenge in extreme-scale systems. Errors, variation, and failure rates are expected to increase in future fabrication technologies, particularly when operated at near-threshold voltages. Moreover, the large scale of exaOP supercomputers will lead to high error and failure rates per system. Finally, the drive for energy-efficiency may lead to systems that operate with smaller guard bands than today's systems, increasing transient error rates.

Extreme-scale systems must tolerate these non-ideal behaviors, and can only afford to devote a small amount of energy to doing so, which prohibits the use of many of the redundancy-based reliability mechanisms that have been used in the past. Runnemede takes a cross-layer [8] approach to reliability that combines hardware-based error detection with software-based recovery and adaptation mechanisms, minimizing reliability overheads during the common case of correct operation. However, we also incorporate hardware-based recovery techniques, such as ECC memory, where they are energy- and complexity-effective. We use a scalable checkpointing approach based on Rebound [2] to protect state against unrecoverable errors. We envision that a supercomputer-scale Runnemede system will use multiple levels of checkpointing, including checkpointing to DRAM, NVRAM, and hard disks, to reduce checkpoint and recovery overheads for common, localized failures, while still protecting itself against uncommon system-wide failures.

4. Sunshine: an Initial Design

Early in the Runnemede project, we began the design of a test chip, code-named "Sunshine." Intended for fabrication in 22nm technology, Sunshine would have been a demonstration of our architecture and a platform for our software team's work. While the Sunshine test chip was never actually built, the process of designing a test chip contributed significantly to the ideas that went into Runnemede.

To strike a balance between utility and implementation effort, Sunshine incorporated most of the programmer-visible features in the Runnemede architecture, such as the scratchpad memories, software-managed caches, register-based power management, and the XE-CE communication interface. However, other aspects of the design, in particular the architecture and microarchitecture of the cores, were chosen to minimize implementation effort.

Sunshine's CEs were based on the Siskiyou Peak [32] synthesizable core, with a custom interface to the on-chip networks and memory hierarchy. The XEs were single-issue in-order cores with a custom RISC ISA that incorporated instructions for software-managed caches, synchronization, network collectives, and block memory transfers. Several Sunshine implementations were considered, including a multi-block chip that supported multi-chip systems.

One area where the Sunshine work heavily influenced the larger Runnemede effort was the sizing of the scratchpads and caches shown in Figure 1. Sunshine's target clock rate was 500 MHz – 1 GHz, and our analysis suggested that 64KB was the largest scratchpad that would fit in that clock cycle when implemented in energy-efficient SRAM. Similarly, the software-managed caches were sized at 32KB because of the latency incurred by tag lookup and hit/miss checks. We selected 2.5MB as the L2 scratchpad size in order to keep the first-level (within a block) network latency under one cycle. The L3 and L4 scratchpad sizes shown in the figure were not directly driven by the Sunshine design.

5. Initial Evaluation

As an initial evaluation of Runnemede, we present three results: a case study showing how hardware-software co-design improves the energy efficiency of an application, an analysis of the energy/bandwidth trade-offs in different network topologies, and a comparison of the energy costs of scratchpad-based and cache-based memory hierarchies.

The co-design results were generated using a functional simulator of the Sunshine architecture, while the network results were generated using an analytic model that accounts for wire length, switch size, and switch energy. Our memory analysis was done with a trace-driven simulator, using traces generated with a custom PIN [26] tool and a set of libraries that allows programmers to write scratchpad-style programs for Linux workstations. Our energy estimates were generated using an internal power model that estimates the energy of each functional block of a design (wires, memories, pipeline stages, etc.) based on representative circuits from existing designs. The unit of energy in our results is the amount of energy required to perform a double-precision floating-point multiply (FM64), which we selected to allow a comparison between different simulations while still being free of fabrication process details.

5.1. HW-SW Co-Design Case Study

One of the five challenge problems used to benchmark UHPC architectures is a streaming sensor application based on synthetic aperture radar (SAR). SAR's input is a set of vectors, each of whose values represent the returns from a given radar pulse as a function of time. Given this set of vectors and the location of the radar at the time each pulse was emitted, SAR generates an output image that shows how much energy was reflected from each point in the image.

We implement our SAR algorithm using the codelet execution model and run it on our simulator, modeling a four-block Sunshine architecture with a total of 32 XEs and 4 CEs. Figure 3 shows how different co-design optimizations reduce the energy consumption of this application. At the top of the figure, the *Base SAR* bar shows the energy consumed by our first implementation of the algorithm, which is dominated by the energy consumed in computation. In particular, SAR performs a large number of `sin` and `cos` operations, and our baseline implementation spends the majority of its time in math library routines.

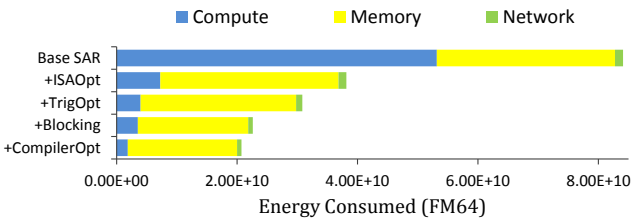

Figure 3. Co-design optimizations for SAR

To address this problem, we add a `sincos` instruction to the ISA that computes both the `sin` and the `cos` of its input (since SAR typically needs both values). The *+ISAOpt* bar on the graph shows the energy consumed with this improvement, which reduces compute energy by 86%. We also implement an algorithmic change that replaces the original single-precision computation of each output pixel's value with a double-precision computation of a subset of the pixels and a less-expensive interpolative computation of the

remaining pixels. This reduces compute energy by an additional 45%, as shown in the *+TrigOpt* bar.

The *+Blocking* bar shows the energy consumed after we modify SAR so that each codelet copies the portions of the input array that it will use into the XE's L1 scratchpad rather than fetching values from DRAM each time they are used. Since pixels that are close together in the output image depend on similar regions of the input pulses, this substantially reduces the number of DRAM references SAR makes.

Finally, our compiler does not perform some address calculation and strength reduction optimizations that a more-mature compiler would perform. Hand-implementing these optimizations reduces computation energy by 47% over the value shown in the *+Blocking* bar, yielding the results shown in the *+CompilerOpt* bar. In total, our co-design optimizations reduce computation energy by 97% and total energy by 75%, showing the value of jointly optimizing hardware, applications, and development tools.

5.1.1. Effect of Technology Scaling

The results presented in Figure 3 assumed a 45nm implementation of Runnemede. Figure 4 shows predictions of how the energy consumed scales with fabrication process (based on internal predictions about process scaling), while Figure 5 shows how the fraction of energy consumed by computation, memory, and the on-chip network scales. As expected, computation energy scales well, decreasing by 77% as we move from 45nm to 10nm, while network energy only decreases by 51%. Memory energy also decreases drastically over time, driven by the energy per byte improvements from the use of stacked DRAM. As a result, SAR's energy consumption is relatively balanced between computation, memory, and network in the 10nm node.

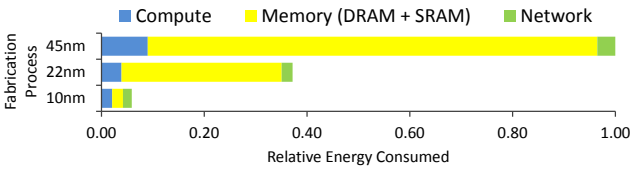

Figure 4. SAR energy scaling

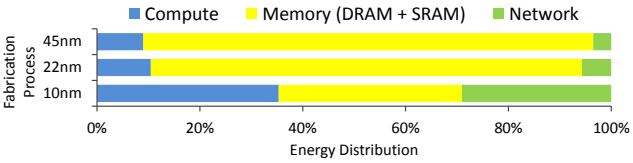

Figure 5. SAR energy distribution

5.2. Network Analysis

Our network analysis focuses on minimizing the amount of energy required to send each message (or packet) in the network. Extreme-scale applications need significant communication locality in order to meet their energy goals, encouraging the design of networks that minimize the cost of short-distance communication even if that somewhat increases the cost of long-distance communication. However, we also have to ensure that the on-chip network provides enough global bandwidth to not limit the performance of application phases that require global communication, under the assumption that such phases power-down other portions of the chip in order to free up power for communication.

Given these guidelines, we examine a number of tree-based networks, as they provide high bandwidth with a small number of switch crossings for local messages. In addition, recent work [23, 22] suggests that designs based on relatively high-radix switches reduce network energy.

Tree-based networks can also provide differing amounts of bandwidth at each level in the tree to tune the ratio of local to global bandwidth. Figure 6 illustrates this effect. In a pruned tree, each link in the network has the same bandwidth. As a result, the total bandwidth at each level in the network decreases by the radix of the switches used. At the other extreme, the bandwidth of the links in a fat tree scales up by the switch radix at each level, keeping the total bandwidth per level constant. In between these extremes, the bandwidth per link of a hybrid tree increases as one moves up the tree, but at a rate smaller than the switch radix, so the global bandwidth at each level decreases more slowly than the global bandwidth of the pruned tree. In our experiments, we consider hybrid tree networks in which the total bandwidth at each level is a factor of two lower than the total bandwidth of the level below it, regardless of switch radix.

We evaluate our networks by calculating the energy per bit of messages in two access patterns: uniform-random and a localized pattern. In the latter, each message's destination is selected randomly from the nodes at a distance H from the source, where the probability that a message travels H hops is $\frac{N}{K_H H}$, with K_H being the number of nodes at distance H from the source node, and N being a normalizing constant selected such that $\sum_{\forall H} \frac{N}{K_H H} = 1$. In our analysis, we model the number of hops each message traverses, the wire length of each hop, and the switch energy of each configuration.

Figure 7 shows the results of our localized-traffic analysis. The fat tree and pruned tree topologies show energy minima between 4-ary and 16-ary trees, depending on the size of the network. These minima are less evident in the hybrid tree topologies, whose energy curves are closer to monotonically-increasing with switch radix.

A Runnemede system fabricated in 2018-2020 would be

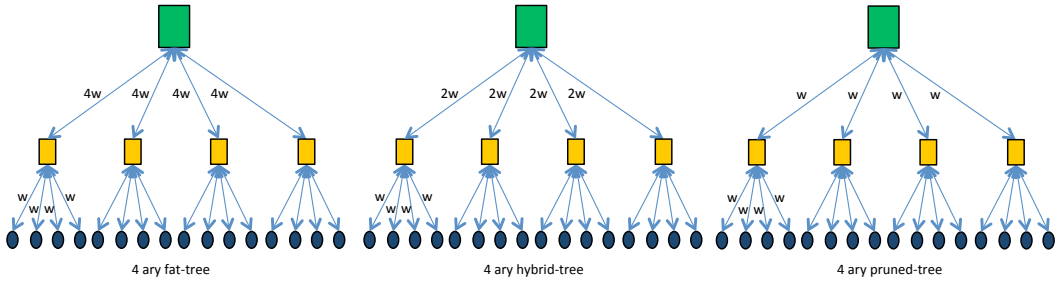

Figure 6. Schematic of a fat tree, a hybrid tree and a pruned tree

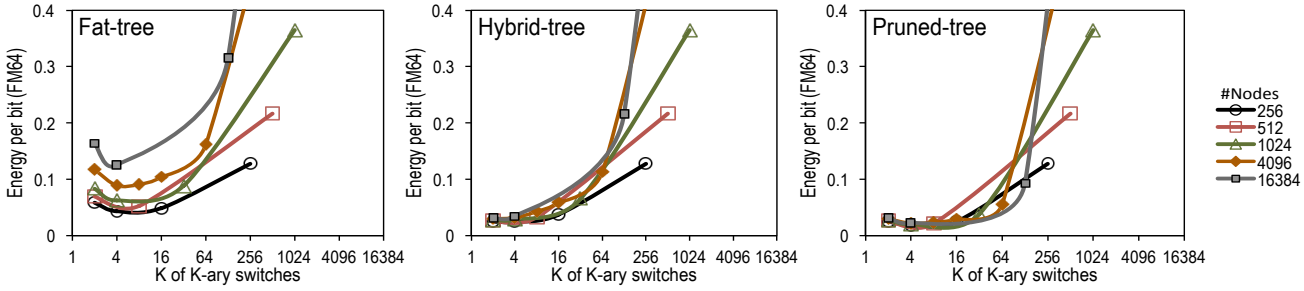

Figure 7. Energy per bit for localized traffic as a function of switch radix

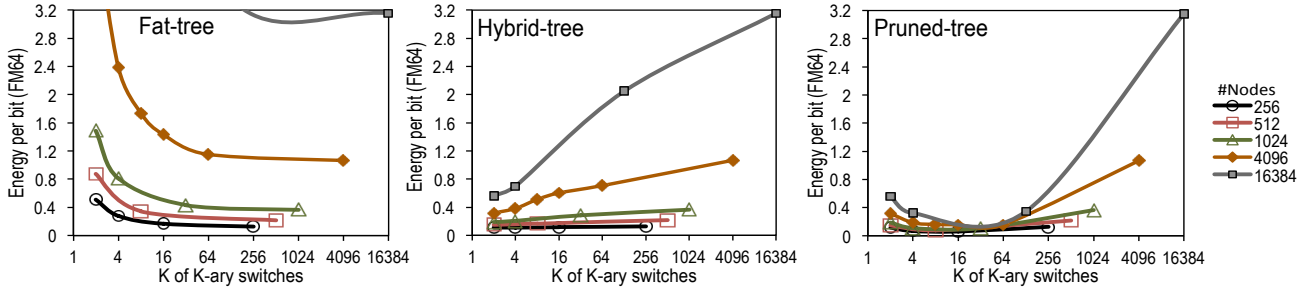

Figure 8. Energy per bit for uniform random traffic as a function of switch radix

expected to have a 512- to 1,024-node on-chip network. Focusing on the 1,024-node network, the curves show that the best configurations of the pruned tree and hybrid tree topologies require very similar amounts of energy to send a message, and that the curve is relatively flat near the optimal point. The 512-node network also sees similar message energy in the hybrid tree and pruned tree topologies, although its slope is distorted because there are only a small number of valid switch radices for 512-node trees if all levels in the tree are required to have the same radix. The fat tree network shows significantly higher message energies for both the 512-node and 1,024-node networks, making it unattractive for Runnemede.

On uniform-random traffic, as shown in Figure 8, the different networks have very different energy trends. In the fat tree network, message energy decreases significantly as ary, and thus switch size, increase, while message energy *increases* monotonically with switch radix in the hybrid tree network. Finally, the pruned tree network sees high message energies for both very small and very large switch sizes, with minima in between.

These curves can be explained by considering the size, and thus the energy cost, of the switches in each network. Fat trees use wider channels in higher levels of the tree, making switches in the top levels large and energy-expensive. This encourages the use of high-radix switches that increase the number of nodes a given node can reach via the lower levels of the tree. In pruned trees, switches at all levels of the network are the same size, creating a trade-off between switch size and the number of switches traversed by an average message and leading to sub-optimal energy at either extreme of radix size. Hybrid trees lie between these two extremes: their link widths increase at higher levels in the tree, but at a much slower rate than fat trees. This causes them to have significantly lower message energies than fat trees, approaching the message energy of pruned trees.

Table 1. Runnemede network parameters

Network	Link Width	Radix (Modules Connected)
First-Level Network	N	11 (1 CE, 8 XEs, 1 L2 Memory, 1 port to Second-Level Network)
Second-Level Network	2N	10 (8 First-Level Network Ports, 1 L3 Memory, 1 Third-Level Network Port)
Third-Level Network	4N	11 (8 Second-Level Network Ports, 1 L4 Memory, 1 Off-Chip Memory, 1 Off-Chip Network)

Based on these results, we envision a hybrid tree network for Runnemede, but one whose link bandwidth increases more slowly with level in the tree than the hybrid tree networks evaluated here. Pruned tree networks had the lowest message energy in our studies, but provide very little bisection bandwidth, making them a performance bottleneck on applications with limited communication locality. Fat tree networks have high bisection bandwidth, but are too energy-expensive. Our results also suggest that an 8-ary or 16-ary tree is close to energy-optimal, leading to a three-level tree for 512- to 1,024-node networks. Based on this analysis, Table 5.2 shows the parameters for the Runnemede network.

5.3. Evaluating Scratchpad Memories

To evaluate the energy-efficiency of scratchpad memories, we simulate the execution of a 1,024x1,024 Givens QR decomposition and a 2,048x2,048 matrix-matrix multiplication on an 8-XE block of Runnemede. We start with a sequential version of each program, and use the R-Stream compiler [27] to generate parallel versions for hardware-coherent caches with different sets of cache locality optimizations. We also modify R-Stream to automatically compile applications to a scratchpad-based memory hierarchy, although, at the moment, the compiler can only target one level of scratchpads. Finally, we hand-code versions of the applications to take advantage of the two-level scratchpad hierarchy present in a Runnemede block. Our results show the active memory energy consumed by each benchmark, neglecting leakage energy because because our trace-driven simulator does not model time accurately.

We model the scratchpad-based memory hierarchy of a single block as shown in Figure 1, with 64KB L1 scratchpads in each core and a 2MB L2 scratchpad (rounding to the nearest power-of-two bytes). We also simulate a memory hierarchy with hardware-coherent 64KB L1 and 2MB L2 caches. Both levels of cache use 64-byte lines and are 8-way set-associative. We model an "oracular" directory-based MESI coherence protocol in which each cache has complete knowledge about the contents of the other caches in the system. These results assume DDR DRAM off-chip memories, because more detailed information about their power consumption is available than for stacked DRAMs.

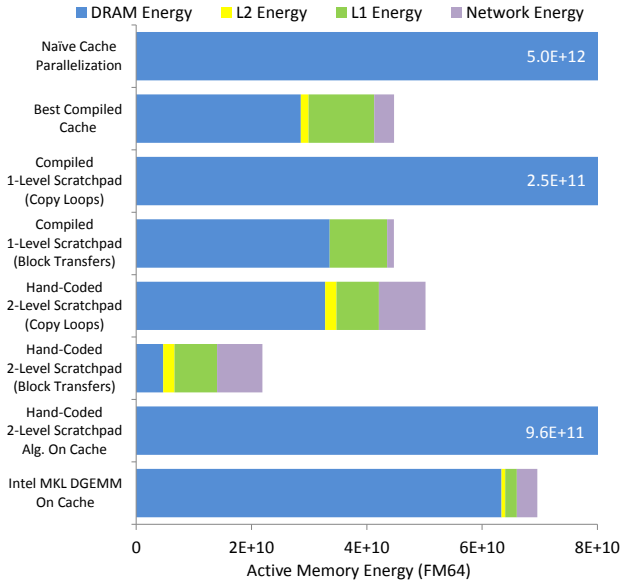

Figure 9. Memory energy for matrix mult.

5.3.1. Matrix Multiplication

Figure 9 shows the results of our matrix multiplication experiments. The *Naïve Cache Parallelization* bar shows the energy consumed in the memory system when executing an 8-threaded matrix multiplication with no blocking for cache locality on our cache hierarchy. The *Best Compiled Cache* bar shows the energy consumed on the same hierarchy when all of R-Stream's cache locality optimizations are applied and the arrays holding each matrix are padded to prevent set conflicts. Applying these optimizations reduces memory system energy by over two orders of magnitude, from 4.96×10^{12} FM64 to 4.47×10^{10} FM64.

Next, we use R-Stream to compile versions of matrix multiplication for our scratchpad-based memory hierarchy, generating variants that use either copy loops or block transfer operations to move data. The *Compiled 1-Level Scratchpad* bars show the energy used by these versions of the computation. When block transfers are used to copy data, R-Stream is able to achieve the same energy consumption as a two-level hardware-coherent cache hierarchy, in spite of only being able to take advantage of the L1 scratchpads. In contrast, the copy loop version of the scratchpad-based algorithm consumes almost 6x more energy than the best cache-based algorithm, due to the 7x increase in energy per byte when performing single-word accesses to the DRAMs.

The *Hand-Coded 2-Level Scratchpad* bars show the energy consumed by a hand-written matrix multiplication that takes advantage of both the L1 and L2 scratchpads in a block. When copy loops are used, the hand-coded computation approaches the energy of the best cache-based code,

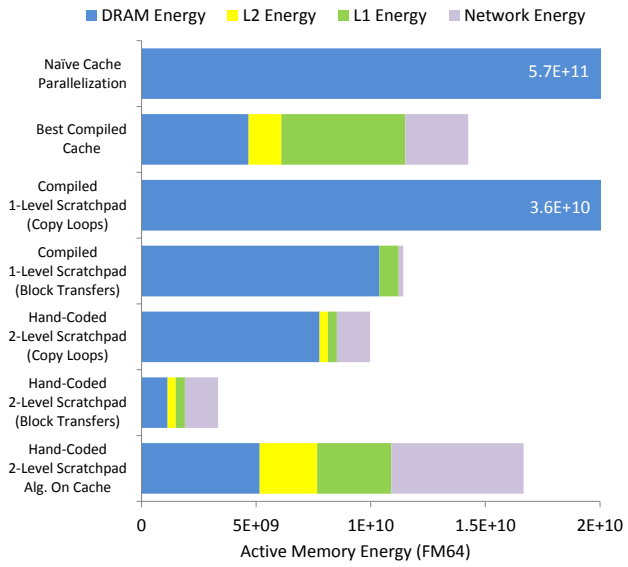

Figure 10. Memory energy for Givens QR

even with the energy/byte penalty of single-word DRAM accesses. When block transfers are used, energy consumption decreases to 49% of the best cache-based code.

The last two bars of the graph are included as sanity checks on our results. The *Hand-Coded 2-Level Scratchpad Alg. on Cache* bar shows the energy consumed when the scratchpad-based code is run on a hardware-coherent cache hierarchy, to demonstrate that our energy efficiency gains are due to scratchpads, not to a more-efficient underlying algorithm. The *Intel MKL DGEMM on Cache* bar shows the memory energy consumed by an eight-threaded 2,048x2048 matrix multiplication using the DGEMM routine from Intel's Math Kernel Libraries [29]. Because the MKL DGEMM was optimized for a different memory hierarchy, it is not possible to directly compare its results to our other results, but the fact that the MKL DGEMM did not beat our compiled codes argues that R-Stream does a good job optimizing for cache locality.

5.3.2. Givens QR

Figure 10 shows the results of our experiments with Givens QR decomposition. Applying locality optimizations to the cache-based code reduces memory energy by 40x over a naïve parallelization. When block transfers are used for data movement, the R-Stream-compiled scratchpad-based code consumes 20% less energy than the best cache-based code, again using only the L1 scratchpads.

The hand-coded version of Givens QR that takes advantage of both the L1 and the L2 scratchpads uses 30% less memory system energy than the best cache-based version, even when energy-inefficient copy loops are used for data movement. When the block transfer operations are used, the scratchpad algorithm uses 76% less memory energy than the cache-based algorithm. Again, we also run the hand-coded scratchpad code on a cache-based hierarchy to show that scratchpads are responsible for the improvement.

These results suggest that, in some cases, giving programs direct control over on-chip memory can significantly reduce memory system energy. This reduction in energy comes at a non-trivial cost in programmer effort, although our results and those of others [25][4][10] suggest that compilers may be able to automate scratchpad management, at least for regular codes.

6. Related Work

Runnemede is one of four extreme-scale architecture research projects funded by the UHPC program. The NVIDIA-led Echelon team [21] developed a GPU-inspired architecture that integrates a large number of throughput-optimized cores and a smaller number of latency-optimized cores onto a chip. Sandia's X-Caliber project combined latency-optimized cores with compute-near-memory units using 3-D stacking, and MIT's Angstrom group [3] explored techniques for self-aware computing systems and factored operating systems.

Our execution model [24] is based on the Codelet paradigm [14] [36], which is separately embodied in ETI's SWARM runtime [11]. Our high-level compiler team is developing tools that use the Hierarchically-Tiled Arrays [13] [5] and Concurrent Collections [7] models to compile conventional programming languages into codelet-based programs. We have also been contributing to the development of the Open Community Runtime [31] and have been working with researchers from Rice University to apply concepts from their Habanero [9] model to our system.

The power-gating, clock-gating, and NTV techniques used in Runnemede build on a large body of circuit research at Intel [20] [1] [16]. Recently, Intel Labs demonstrated an experimental NTV IA-32 processor, code-named Claremont [18], that achieves a 4.7x increase in energy efficiency by reducing its supply voltage from 1.2V to 0.45V.

7. Conclusions

Runnemede is a "blank sheet of paper" research architecture designed to maximize energy efficiency without the constraints imposed by backward compatibility and the need to support conventional programming models. Runnemede's focus on energy-efficiency begins at the circuit level, using NTV circuits and fine-grained power and clock gating to minimize power dissipation. At the architectural level, we use many simple cores, organize them into

hierarchical groups, and divide them into Execution Engines and Control Engines to allow separate optimization of hardware for OS code and application kernels. Our memory system emphasizes efficiency through a single address space, software-managed scratchpads, incoherent caches, and block operations. Our on-chip network is hierarchical and provides support for barriers and collectives.

We continue to develop Runnemede through a co-design process that simultaneously explores hardware architectures, runtime/OS mechanisms, and applications. Our initial experience with Runnemede has shown that co-design can significantly reduce application energy and has demonstrated the potential of application-managed memory hierarchies. However, many questions remain unanswered about programmability, execution engine architecture, and off-chip networking, to name a few. Our ongoing work is exploring these issues, and the Runnemede architecture will continue to evolve as this work proceeds.

8. Acknowledgments

This research was, in part, funded by the U.S. Government. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing the official policies, either expressed or implied, of the U.S. Government, the Intel Corporation, the University of Illinois at Urbana-Champaign, or Reservoir Labs, Inc. We would like to thank the anonymous reviewers, Doug Carmean, Jim Held, Justin Rattner, and Terry Smith for their feedback about early versions of this paper.

References

[1] A. Agarwal et al. A 320mV-to-1.2V on-die fine-grained reconfigurable fabric for DSP/media accelerators in 32nm CMOS. In *ISSCC*, 2010.

[2] R. Agarwal, P. Garg, and J. Torrellas. Rebound: Scalable checkpointing for coherent shared memory. In *ISCA*, 2011.

[3] The MIT angstrom project. `projects.csail.mit.edu/angstrom`.

[4] O. Avissar, R. Barua, and D. Stewart. An optimal memory allocation scheme for scratch-pad-based embedded systems. *ACM Transactions on Embedded Computing Systems*, 1(1):6–26, Nov. 2002.

[5] G. Bikshandi et al. Programming for parallelism and locality with hierarchically tiled arrays. In *PPOPP*, 2006.

[6] S. Borkar and A. A. Chien. The future of microprocessors. *Communications of the ACM*, 54(5):67–77, 2011.

[7] Z. Budimlic et al. Concurrent collections. *Scientific Programming*, (18), 2010.

[8] N. P. Carter, H. Naeimi, and D. S. Gardner. Design techniques for cross-layer resilience. In *DATE*, 2010.

[9] V. Cave, J. Zhao, J. Shirako, and V. Sarkar. Habanero-java: the new adventures of old X10. In *Principles and Practice of Programming in Java*, 2011.

[10] A. Dominguez, S. Udayakumaran, and R. Barua. Heap data allocation to scratch-pad memory in embedded systems. *Journal of Embedded Comptuing*, 1(4):521–540, 2005.

[11] ET International. SWARM (swift adaptive runtime machine). Technical report, ET International, 2011.

[12] M. Fillo et al. The M-Machine multicomputer. In *MICRO*, 1995.

[13] B. B. Fraguela et al. The hierarchically tiled arrays programming approach. In *LCR*, Houston, Texas, 2004.

[14] G. R. Gao, J. Suetterlein, and S. Zuckerman. Toward an execution model for extreme-scale systems–runnemede and beyond. Technical Report CAPSL TR 104, University of Delaware, April 2011.

[15] R. Hameed et al. Understanding sources of inefficiency in general-purpose chips. In *ISCA*, 2010.

[16] S. Hsu et al. A 280mV-to-1.1V 256b reconfigurable SIMD vector permutation engine with 2-dimensional shuffle in 22nm CMOS. In *ISSCC*, 2012.

[17] Intel Corporation. Intel® turbo boost technology in Intel® Core™ microarchitecture (Nehalem) based processors. White Paper, November 2008.

[18] S. Jain et al. A 280mV-to-1.2V wide-operating-range IA-32 processor in 32nm CMOS. In *ISSCC*, pages 202–203, 2012.

[19] U. R. Karpuzcu, K. B. Kolluru, N. S. Kim, and J. Torrellas. VARIUS-NTV: A microarchitectural model to capture the increased sensitivity of manycores to process variations at near-threshold voltages. In *DSN*, 2012.

[20] H. Kaul et al. A 300mV 494GOPS/W reconfigurable dual-supply 4-way SIMD vector processing accelerator in 45nm CMOS. In *ISSCC*, 2009.

[21] S. W. Keckler et al. GPUs and the future of parallel computing. *IEEE Micro*, 31(5):7–17, 2011.

[22] J. Kim, W. Dally, S. Scott, and D. Abts. Technology-driven, highly-scalable dragonfly topology. In *ISCA*, 2008.

[23] J. Kim, W. J. Dally, and D. Abts. Flattened butterfly: a cost-efficient topology for high-radix networks. In *ISCA*, 2007.

[24] R. Knauerhase et al. For extreme parallelism, your OS is Sooooo last-millennium. In *HotPar*, 2012.

[25] T. Knight, J. Park, M. Ren, and M. Houston. Compilation for explicitly managed memory hierarchies. In *PPOPP*, 2007.

[26] C.-K. Luk et al. Pin: Building customized program analysis tools with dynamic instrumentation. In *PLDI*, 2005.

[27] B. Meister et al. R-stream compiler. In *Encyclopedia of Parallel Computing*. Springer Reference, 2011.

[28] Micron Technology, Inc. Hybrid memory cube. http://www.micron.com/products/hybrid-memory-cube, 2011.

[29] Intel math kernel library. `http://software.intel.com/en-us/intel-mkl`.

[30] W. A. Najjar, E. A. Lee, and G. R. Gao. Advances in the dataflow computational model. *Parallel Computing*, 1999.

[31] Building an open community runtime (OCR) framework for exascale systems. `http://sc12.supercomputing.org/schedule/event_detail.php?evid=bof219`.

[32] J. Rattner. Extreme scale computing. ISCA Keynote, 2012.

[33] S. L. Scott. Synchronization and communication in the T3E multiprocessor. In *ASPLOS*, 1996.

[34] Ubiquitous high performance computing (UHPC). `http://www.darpa.mil/Our_Work/MTO/Programs/Ubiquitous_High_Performance_Computing_(UHPC).aspx`.

[35] G. Venkatesh et al. Conservation cores: Reducing the energy of mature computations. In *ASPLOS*, 2010.

[36] S. Zuckerman, J. Suetterlein, R. Knauerhase, and G. R. Gao. Position paper: Using a codelet program execution model for exascale machines. In *EXADAPT Workshop*, 2011.