Streamware: Programming General-Purpose Multicore Processors Using Streams

Jayanth Gummaraju‡ Joel Coburn‡ Yoshio Turner§ Mendel Rosenblum‡

‡Computer Systems Laboratory, Stanford University, Stanford, CA 94305

§Hewlett-Packard Laboratories, Palo Alto, CA 94304

Abstract

Recently, the number of cores on general-purpose processors has been increasing rapidly. Using conventional programming models, it is challenging to effectively exploit these cores for maximal performance. An interesting alternative candidate for programming multiple cores is the stream programming model, which provides a framework for writing programs in a sequential-style while greatly simplifying the task of automatic parallelization. It has been shown that not only traditional media/image applications but also more general-purpose data-intensive applications can be expressed in the stream programming style.

In this paper, we investigate the potential to use the stream programming model to efficiently utilize commodity multicore general-purpose processors (e.g., Intel/AMD). Although several stream languages and stream compilers have recently been developed, they typically target special-purpose stream processors. In contrast, we propose a flexible software system, Streamware, which automatically maps stream programs onto a wide variety of general-purpose multicore processor configurations. We leverage existing compilation framework for stream processors and design a runtime environment which takes as input the output of these stream compilers in the form of machine-independent stream virtual machine code. The runtime environment assigns work to processor cores considering processor/cache configurations and adapts to workload variations. We evaluate this approach for a few general-purpose scientific applications on real hardware and a cycle-level simulator set-up to showcase scaling and contention issues. The results show that the stream programming model is a good choice for efficiently exploiting modern and future multicore CPUs for an important class of applications.

Categories and Subject Descriptors D.1.3 [*Programming Techniques*]: Concurrent Programming (Parallel Programming); C.4 [*Performance of Systems*]: Design Studies

General Terms Design, Experimentation, Languages, Performance

Keywords Streams, General-Purpose Multicore Processors, Programming, Runtime System

ASPLOS '08, March 1–5, 2008, Seattle, Washington, USA.

1. Introduction

Over the last couple of years the general-purpose processor industry has undergone a fundamental transformation. We have entered the multicore era where the increasing number of on-chip transistors is used for placing more cores on chip rather than for designing faster, more complex single-core processors. Quad-core chips are already in the market and the number of on-chip cores is expected to reach as many as 16 by the end of the decade.

Developing software to fully exploit the many cores for maximal performance, however, has become a big challenge. Using conventional programming models (e.g., multi-threaded programming), developers spend considerable effort trying to write parallel programs, yet problems stemming from synchronization, scheduling and communication have limited the success of this approach. It is widely agreed that parallel programming is hard and the average programmer is going to need significant help in efficiently mapping one's code to modern multicore processors.

This challenge is being exacerbated by the new generations of multicore processors with varying numbers of cores and cache memory hierarchies that make writing portable parallel programs even more difficult. It is not clear how to write a program or library that can efficiently map onto multiple generations of multicore processors from different vendors.

In this paper we discuss the potential to use the stream programming model [25, 37] to fully exploit general-purpose multicore processors. Stream programs use a different programming style (i.e., data-flow) from traditional programming models (i.e., von Neumann). To get performance benefits, stream programming involves bulk loading of data into a local memory (LM), operating on the data in parallel, and bulk storing of the data back into memory.

Stream programing has been the programming model of choice for special-purpose stream processors like Cell, Imagine, and Merrimac [20, 25, 13], which contain several cores. This model follows a data-flow framework, where the computation and communication are separated and made explicit by the programmer. This style of programming has not only been used to write image/video processing applications, but has evolved into a more general-purpose model encompassing a broader set of data-intensive applications including irregular scientific applications [28].

We argue that coding programs in a stream style allows them to be written sequentially and can be automatically parallelized and efficiently mapped onto multiple processor cores. The stream programming model frees the programmer from worrying about most of the hard problems of parallel programming including spawning and scheduling multiple threads of control, synchronization and communication issues, and portability across different hardware configurations.

We present Streamware, a software system which enables these stream programs to be automatically mapped to a wide variety of general-purpose multicore processor configurations. We leverage existing compilation frameworks for stream processors that can compile stream programs down to a common virtual machine code (i.e., Stream Virtual Machine or SVM) [26]. This compilation process is based on well-understood and commonly deployed compiler techniques [16, 14, 5, 29, 34]. We design and implement a runtime system which exports an API that is targeted by such higher level compilers and automatically maps to the underlying processor.

Our Streamware runtime system performs cache hierarchy aware dynamic scheduling for mapping efficiently to general-purpose multicore processors. The scheduler ensures that the computation kernels and data are co-located in such a way that the kernel executes on the core closest to the cache that contains the data in the cache hierarchy. This enables the data to be efficiently fed to the functional units present in the cores (e.g., short-vector SIMD units such as the SSE units of x86 processors). The scheduler is also dynamic so it can handle large variance in execution times (e.g., due to heterogeneous cores) as well as react to other processes running on the machine. Using real machines and simulations we show the efficiency of the mapping on a range of workloads and evaluate the scalability up to 16 cores.

The rest of the paper is organized as follows. Section 2 describes the stream programming model and its essential attributes. Section 3 presents the design of our dynamic, cache hierarchy aware runtime system which automatically maps SVM code to various hardware configurations. Section 4 presents the experimental performance evaluation on a real multicore hardware and on a cycle-accurate simulation system. Section 5 discusses related work, and Section 6 concludes the paper.

2. Stream Programming and Applications

Stream programming was originally developed for media and image processing applications with simple, regular data access patterns and statically predictable control. More recently it has grown into a more general-purpose programming model successfully applied to scientific applications and other irregular applications [28]. In this section we try to separate out the essential components of stream programming and highlight the core attributes useful for developing a stream software environment for general-purpose processors.

2.1 Stream Programming Model

Stream programming advocates a style of programming where data is encapsulated into contiguous array of records called *streams* which get operated on by a series of computation *kernels*. Figure 1 illustrates how streams and kernels are choreographed to execute in synch. Essentially, the program is structured in a data-flow style at the granularity of streams and kernels.

This style of programming makes explicit the *parallelism* and *locality* present in applications – *parallelism* is encoded within computation kernels and the *locality* of data is ensured both spatially (contiguous chunks – streams) and temporally (producer - consumer locality). If kernels are data-parallel, they can be broken down into mutually independent chunks which can be executed in parallel.

Although many applications can potentially be expressed using the stream programming style, some of the desirable application characteristics include: large amounts of data to operate on, high arithmetic intensity, memory accesses that can be determined well in advance of their use, and producer-consumer locality between computation kernels.

Initially targeted only for media/regular applications, stream programming has evolved into a more general programming

```
for (ns = 0; ns < NUM_STRIPS; ns += 2) {
  for (i = 0; i < 2; i++ ) { //Double buffering

    streamLoad (as_i, a, start_idx, N_s, sizeof (Type a));

    streamLoad (idxs_i, idx, start_idx, N_s, sizeof(int));
    addDep (Gcs_i , Lidxs_i );
    streamGather (cs_i, c, idxs_i, N_s, sizeof (Type c));

    addDep (K_1 , Las_i, Lidxs_i, Gcs_i);
    kernelCall (K_1, as_i, cs_i, ds_i);

    streamLoad (xs_i, x, start_idx, N_s, sizeof (Type x));

    addDep (K_2, K_1, Las_i, Lxs_i);  //K_2 depends on K_1, loads to as_i and xs_i
    kernelCall (K_2, ds_i, as_i, xs_i, zs_i);

    addDep (Szs_i, K_2);
    streamStore (zs_i, z, start_idx, N_s, sizeof (Type z));

    start_idx += N_s;
  }
}
```

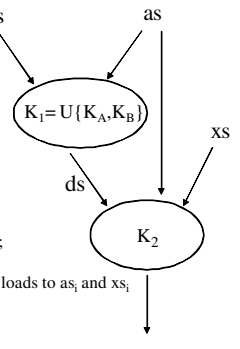

Figure 1. Example of SVM-C pseudo-code and stream graph. Kernel K_1, which is a fusion of kernels K_A and K_B of the high level stream program, takes in as input streams as and cs, and produces stream ds. K_2 takes the output of K_1 (ds), as and xs, and produces zs.

paradigm. Many applications in SpecFP 2006 and Berkeley dwarfs have been shown to fit in this framework [28, 13, 18] (e.g., fluid dynamics, molecular dynamics, structural mechanics, sparse linear algebra, image processing).

The stream programming model advocates a *gather–compute–scatter* style of programming[1]. Data is *gathered* in bulk from arbitrary memory locations in main memory into a local memory (LM). This involves an asynchronous *copy* of data from the main memory address space to the LM address space. Computation *kernels* directly operate on the stream data from LM and the *produced* results are stored back into the LM. The *consumer* kernels use these results during execution and store their results back to LM, and so on. Finally, only the live data from the LM is *scattered* back in bulk to main memory.

Unlike in the traditional thread-based models, the computation and memory operations are explicit and decoupled in the stream model. The memory operations regularize irregular memory accesses by gathering data from different portions of memory and copying them into contiguous addresses in local memory. This enables the computation kernels to perform mainly sequential, local memory accesses and can therefore, run very efficiently (e.g., using SSE instructions in x86). Furthermore, the emphasis is on data placement and co-locating work (i.e., computation) with the already allocated data, unlike traditional thread-based models where data is brought to the site of work.

Several stream languages have been developed over the past few years [11, 37, 22]. However, as depicted in Figure 2, all these languages can be mapped to a common stream virtual machine abstraction layer (SVM) similar to the work discussed in [26]. The SVM code is comprised of DMA operations and computation kernels operating on strips of streams, and the dependencies between them. Finally, the SVM code is compiled to the underlying stream processor using a processor specific compiler.

In order to map a stream program onto SVM code several simple transformations are performed by a stream compiler (Figure 2). Streams are broken down into *strips*, each typically several thou-

[1] Only the style of programming and execution changes. We can continue to use existing sequential languages (e.g., C, Fortran) with a few additional library calls for bulk memory operations.

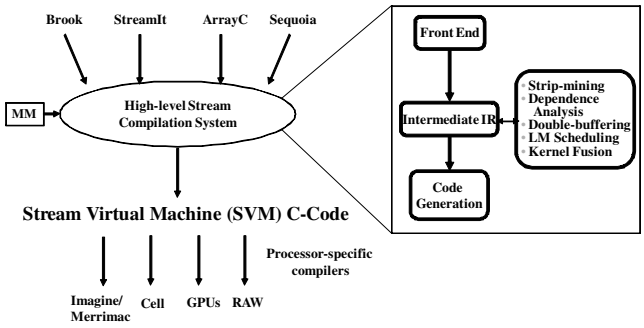

Figure 2. Stream programming platform

sand bytes long, to insure that the working set of strips fits in the LM. The strips are *double buffered* so that when one buffer is being loaded from memory, the other (already loaded) buffer can be operated upon in parallel by the computation kernels. The compiler also inserts synchronization routines between asynchronous bulk memory operations/kernels (Figure 1). Since stream processors impose several resource constraints (e.g., fixed LM size, local registers, etc) the machine model (MM) of the underlying stream processor is input to the stream compiler for maximal performance.

While all stream programming languages must be able to be mapped to the stream virtual machine abstraction, they may express differently the basic stream programming semantics discussed above and may impose different restrictions.

2.2 Generalizations on the Stream Programming Model

In this section we define the essential attributes of a generalized stream programming model which captures the fundamental properties of existing stream languages depicted in Figure 2. Because the term *stream programming* has been overloaded and used somewhat differently by different researchers, we attempt to develop a common ground to define its vital attributes, which we assume throughout the paper. The idea is to decouple the stream programming model from the restrictive constraints imposed by the stream architectures in order to map a wider variety of stream applications to general-purpose architectures.

Generalized Computation Kernels: The computation kernels can contain variable amounts of work per invocation, depending on the control flow inside the kernel. Kernels can be stateful (i.e., carry state between different invocations of the same kernel) or stateless (e.g., data-parallel kernels). Examples of stateful kernels range from simple *reduction variables* (e.g., summation over all elements of the stream) to a more general case where arbitrary amounts of state are carried across invocations.

Variable input/output rates of streams between kernels: The number of input/output elements consumed/produced by the kernel could vary across kernel invocations. For example, the number of output stream elements produced could be data-dependent, resulting in an arbitrary number of elements produced for every kernel invocation.

Synchronization: Synchronization between kernels/bulk memory operations/scalar variables can be implicit or explicit in the stream programming model. However, the stream compiler must explicitly insert dependencies while generating the SVM code using dependence analyses. Typical synchronization points include: boundaries for kernels that perform scalar reductions, before an indexed gather operation for which the index stream is an output of a previous kernel, and at the end of all kernels before executing the dependent scalar code (e.g., writing the final output to a file).

Communication: Communication with the memory system and between memory and kernel operations could be explicit or implicit in the programming model. For example, Accelerator [36] derives the stream data flow graph automatically from sequential code. However, communication is explicit in the SVM layer. Communication between kernel operations is usually accomplished through a producer-consumer relationship (one kernel produces a stream that another kernel consumes). Multiple invocations of the same kernels also communicate using state variables (e.g., reduction variables).

Weak Memory Consistency model: Accesses to memory can be in any order and are constrained only by the synchronization points described above. Bulk memory operations can proceed out-of-order or even be interspersed. Memory accesses within a bulk memory operation can execute in any order, unless specified otherwise by the programmer. Furthermore, the order of execution of the memory instructions as seen by each core could be different as long as the dependencies, which are indicated by synchronization points, is respected.

Exception Handling: Exceptions are simply handled using the mechanism provided by the language in which the stream program is written. However, additional error handling support can be added at the granularity of kernel invocations/bulk memory operations.

Once a stream program is compiled to SVM code, there are three options to schedule on underlying cores – time-multiplexing [13, 20], space-multiplexing [37], and a hybrid space-time multiplexing [16]. In time-multiplexed stream scheduling every kernel is broken down into mutually independent kernel invocations which execute in parallel on all the underlying cores whereas in space-multiplexed stream scheduling each computation kernel is allocated to a particular core and exploits *pipeline parallelism* between cores for performance. Hybrid space-time multiplexing combines both space- and time- multiplexing where a kernel is multiplexed both spatially (i.e., assigned to particular cores) and temporally (i.e., execute on multiple cores in parallel).

3. Streamware Software Environment for General-Purpose Multicore Processors

In this section we present the design of the Streamware software environment for stream program execution on general-purpose multi-core processors. Streamware provides a platform-independent stream virtual machine abstraction to stream compilers and applications and maps it efficiently to the hardware at run-time. To achieve high utilization of hardware functional units, Streamware implements a run-time system which performs cache hierarchy aware dynamic scheduling of memory operations and kernel operations. The run-time system has low overhead and is portable across diverse multi-core processor configurations, delivering maximal performance limited only by the memory bandwidth and/or computation resources of the hardware.

Streamware uses a time-multiplexing approach applicable to stateless kernels and simple reductions. We focus on these scenarios as they provide the greatest opportunity to exploit large-scale multicore processors. As we describe in Section 3.3, our environment could be extended to support general stateful kernels that require space-multiplexing which can be viewed as a special-case version of time-multiplexed stream scheduling.

3.1 Streamware API

Streamware exposes an application programming interface which provides a platform-independent stream virtual machine abstraction to applications and stream compilers. Loosely inspired by the SVM definition of Labonte et al. [26], the Streamware API (Table 1) is distinguished by its support for features useful for mapping onto diverse multi-core hardware. In particular, the virtual machine model presents a set of Local Memories (LMs) for storing stream data, and a set of Kernel Processors for executing kernels. The Ker-

Table 1. Streamware API (Key Features)

	Function	Description
Scheduling	`svmInit()`	Initialize the Streamware virtual machine (bring LM to cache, initialize queues)
	`svmExit()`	Exit the Streamware virtual machine
	`addDep(int op_id, op_t optyp, unsigned long long kern_dep_vec, unsigned long long mem_dep_vec, ndma)`	Add kernel and/or memory dependencies for the kernel or memory operation ($optyp == KNLOP$ or $MEMOP$) with identifier op_id for LM $ndma$
	`executeWork()`	Execute enqueued kernels and memory operations until all operations for some LM complete (return value is list of completed LMs)
	`Barrier()`	Wait for all pending operations for all LMs to complete
Kernel and Memory Operations	`kernelCall(funcPtr kerName, kernel_start_offet, num_elmts, ndma, streamPtrs str[])`	Enqueue a kernel invocation
	`streamLoad(void *str, void *arr, int num_dims, int start_offset[DIMS], int num_elmts[DIMS], int stride[DIMS], int arr_elmt_size, int ndma)`	Enqueue strided memory copy (e.g., $str[i] = arr[start\_offset + stride * i]$) from uni/multi-dimensional array of records to stream
	`streamStore(void *str, void *arr, int num_dims, int start_offset[DIMS], int num_elmts[DIMS], int stride[DIMS], int arr_elmt_size, int ndma)`	Enqueue strided memory copy (e.g., $arr[start\_offset + stride * i] = str[i]$) from stream to uni/multi-dimensional array of records
	`streamGather(void *str, void *arr, int *index_stream, int num_dims, int start_offset[DIMS], int num_elmts[DIMS], int arr_elmt_size, int ndma)`	Enqueue memory gather (e.g., $str[i] = arr[index\_stream[start\_offset + i]]$ from uni/multi-dimensional array of records to stream
	`streamScatter(void *str, void *arr, int *index_stream, int num_dims, int start_offset[DIMS], int num_elmts[DIMS], int arr_elmt_size, int ndma)`	Enqueue memory scatter (e.g., $arr[index\_stream[start\_offset + i]] = str[i]$ from stream to uni/multi-dimensional array of records
	`streamScatter`**`OP`**`(void *str, void *arr, int *index_stream, int num_dims, int start_offset[DIMS], int num_elmts[DIMS], int arr_elmt_size, int ndma)`	Enqueue memory scatter-op (e.g., if **`OP`** is **`Add`**, $arr[index\_stream[start\_offset + i]] += str[i]$) from stream to uni/multi-dimensional array of records

nel Processors are partitioned into groups that share the same LM. For maximal performance, each Kernel Processor only executes kernels that access stream data in its associated LM.

In contrast to previous SVM APIs which tightly couple the SVM model with the physical machine attributes, Streamware decouples these across the stream compiler and run-time to provide portability to a wide variety of general-purpose processors taking advantage of their more relaxed resource constraints compared to stream processors. This is accomplished by making the physical machine model (MM in Figure 2) an input to the run-time layer instead of the compiler, and by exporting a parameterized stream virtual machine model to the compiler. The compiler takes stream programs written in a high-level language and generates low-level Streamware API code (similar to Figure 2) comprising of bulk memory operators, kernels, and dependencies between them. The compiler performs widely used transformations including strip-mining, double-buffering, kernel fusion, etc and represents strip-sizes, etc as functions of run-time parameters. The run-time environment implements the low-level API and assigns values to these parameters based on the architecture on which the program executes.

Applications using the Streamware run-time are structured as a single control thread consistent with the stream programming style. The control thread calls the Streamware API to submit memory and kernel operations to the run-time environment to be scheduled for execution. After submitting operations, the control thread calls the `executeWork` routine to invoke the run-time to initiate actual processing of submitted operations.

The control thread declares stream objects, which reside in a separate namespace from non-stream scalar data. The API provides functions (`streamLoad`, `streamStore`, etc.) used to submit memory operations that copy data in bulk between the two namespaces. When submitting a stream memory operation, the control thread passes in the ID ($ndma$) of the particular LM that should be the source or destination of the stream data. In addition, the stream memory operations are passed the start address of the stream in LM (str), start address of the data array (arr), start offset to the array, element size, and number of elements. For scatter/gather operations the location in LM of the index stream is also provided. When submitting a kernel operation, the control thread indicates the group of Kernel Processors that can execute the kernel by passing in the ID ($ndma$) of their shared LM. In addition, the `kernelCall` routine is passed the kernel function pointer, start addresses in LM of input and output streams (str), start offsets into the streams ($kernel\_start\_offset$), and number of elements to process (num_elmts). Additionally, the control thread uses the `addDep` call in the API to specify scheduling dependencies between operations, where dependencies are encoded as bitmaps.

It is desirable for the run-time environment to enable applications to run efficiently on diverse multi-core processor implementations without requiring the application to be modified for each target platform. The run-time environment takes into account the specific target processor configuration: the number and types of cores (e.g., single- or multi-threaded cores, homogeneous or heterogeneous cores), and the cache hierarchy (e.g., levels, capacities, and core sharing). The run-time environment abstracts this information and presents it to applications by exporting the following stream virtual machine parameters which get bound to values at run-time: NUM_DMAS (the number of LMs), $NPROC[ndma]$ (the number of Kernel Processors sharing the LM that has ID $ndma$), and $LM\_SIZE[ndma]$ (capacity in bytes of the LM that has ID $ndma$), used by the stream compiler to determine strip sizes). The control thread uses these parameters to calculate loop iteration limits and the values of derived parameters, such as the number of elements to retrieve in each bulk memory operation (i.e., actual strip sizes). Additionally, the control thread uses the parameters to assign work to each LM, ensuring that kernels execute efficiently by accessing stream data from the local LM.

3.2 Mapping to the Hardware

The run-time system maps the Kernel Processor and Local Memory components of the stream virtual machine abstraction to the phys-

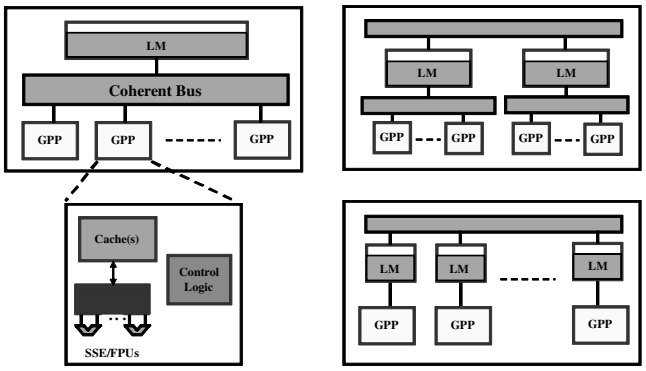

Figure 3. Example multicore configurations

ical multi-core hardware. For each Kernel Processor, the run-time environment spawns a software thread and pins it to a unique physical hardware context (i.e., a processor core or a hardware thread). Each of these software threads is associated with a stream LM and can execute any of the kernels submitted by the control thread for that LM.

Stream data used by the computation kernels are staged in the cache hierarchy by mapping each stream LM to a memory region that is loaded and effectively locked in cache memory throughout program execution [18, 21]. With this approach, most of the associative *ways* in the n-way cache hosting the LM are used for stream data, and the remaining ways are used for kernel instructions and OS data. Previous research efforts have demonstrated that this can be accomplished by using non-temporal loads and stores to access non-stream data, or by manipulating cache line control bits [18, 17].

The memory region dedicated to the LM can be allocated address space arbitrarily larger than the cache capacity. In most cases the application uses the strip sizes determined at run-time (Section 3.1) to limit its LM address space usage to the portion that fits in cache. However, if an application has kernels that can produce or consume data at unpredictable rates, LM addresses outside the range that fits in the cache may be accessed. This is handled transparently by the cache memory system of the general-purpose processor, resulting in cache misses but without compromising application correctness. This constitutes a significant advantage over typical stream processors which have LMs with fixed capacity and would require the addition of special-case LM overflow detection and recovery mechanisms (e.g., kernel suspend/resume) to guarantee correct execution.

Figure 3 shows examples of mapping the LM to multi-core processors with various cache organizations. The cache memories to use as LMs are chosen based on the requirements of relatively large size, high associativity, and a high bandwidth path between the cache and the processor cores. Typically these requirements lead us to choose the last-level on-chip cache. For example, in the configuration on the left in Figure 3 we map the LM to the L2 cache. In this case, the L1 cache serves largely as an extended register file for storing intermediate results during kernel execution [2]. An L3 cache, if present, acts as a bandwidth multiplier by serving as a cache for the LM mapped to L2. Multiple LMs may be mapped to a single cache, potentially improving performance by reducing scheduling contention among cores (we evaluate scaling and contention issues in Section 4.3).

[2] Kernels usually comprise several hundred instructions generating many intermediate results. General-purpose processors typically have few registers to hold these results.

The run-time system implements direct memory access (DMA) engines to perform the asynchronous bulk memory operations submitted by the control thread for each LM. These memory transfers copy data between main memory and LMs at full memory bandwidth. How the run-time environment accomplishes this in general depends on the capabilities of a particular hardware platform. For example, it may create additional software threads (and perhaps bind them to dedicated hardware threads (SMT)) for performing all the bulk memory operations [18]. Alternatively, the Kernel Processor threads could be charged with performing both kernel operations and memory operations. If available, physical DMA engines could be used instead, avoiding software overheads and potentially realizing full overlap of memory and kernel operations [17].

The DMA engines implemented by the runtime are tailored to exploit performance-enhancing features specific to a target processor like software prefetching, quad-word memory copies using short-vector SIMD units (e.g., Intel SSE), hardware performance counters, etc. The implementation also factors in memory system characteristics (e.g., the processor load/store queue size) and datapaths to functional units (e.g., if memory accesses can bypass first-level caches). When performing stream memory operations, the runtime can pack and align stream data in preparation for efficient kernel processing using short vector SIMD units. With the knowledge that all stream data is automatically in packed format, an additional stream compiler pass inserts SSE intrinsic calls in kernels to produce vectorized code.

3.3 Cache Hierarchy Aware Dynamic Scheduling

The runtime environment and the application control thread work in concert to schedule kernel and memory operations. The scheduling policy is cache hierarchy aware, coordinating the scheduling of kernel operations and memory operation to ensure that the stream data accessed by kernels is available in the closest LM in the cache hierarchy. The scheduling policy is also dynamic, assigning work to cores in a flexible manner that adapts to variations in execution times due to workload variation, heterogeneous processor cores, or interference from competing jobs.

The runtime system allocates two *work queues* per LM – one work queue for memory/DMA operations and one work queue for computation kernels. The work queues are filled by the application control thread calling the kernel and memory operations provided by the Streamware API (Table 1). Each call enqueues a descriptor of a kernel or memory operation to the appropriate work queue. Operations in the kernel work queue for an LM are dequeued and executed by the software threads that implement the Kernel Processors that share the LM. Similarly, the DMA engine for the LM dequeues entries from the memory work queue and executes the specified bulk memory operations.

The control thread makes API calls (`addDep`) to specify the dependencies between operations. Dependencies are recorded in the work queues using two bit vectors per entry, one to indicate dependencies on other memory operations, and one to indicate dependencies on other kernel operations. The Kernel Processors and DMA engines respect the dependencies and update the completion bit vectors after performing each operation.

Using the $LM\_SIZE[ndma]$ runtime parameters, the stream compiler computes the strip sizes that will accommodate the working set of streams and generates parameterized code that indicates how streams are divided up into strips. For example, a stream is loaded into a LM one strip at a time by making multiple calls to `streamLoad`. Similarly, multiple kernel invocations process stream data in strip-wise chunks. The application control thread further subdivides each strip into $NPROC[ndma]$ sub-strips and splits each kernel invocation into $NPROC[ndma]$ kernel calls that operate on different sub-strips. The $NPROC[ndma]$ Kernel Pro-

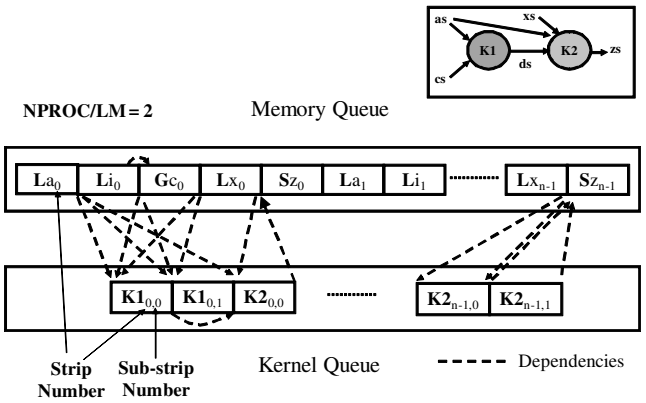

Figure 4. Work Queues for LM. L=Load, G=Gather, S=Store. Elements in the memory and compute work-queues and the dependencies between them for the example stream program presented in Figure 1.

cessors sharing the LM with ID $ndma$ achieve parallel execution by dynamically dequeuing and executing the $NPROC[ndma]$ kernel operations from the kernel work queue. We chose to have kernels operate on sub-strips to allow the DMA operations to use large strip sizes (and hence, perform large bulk transfers) regardless of the number of processors sharing an LM.

To overlap memory operations and kernel executions, the control thread achieves double- or multi-buffering by allocating LM space for multiple strips of each stream and enqueuing operations that access the buffers in round-robin order. The control thread executes a loop which enqueues the operations for the multiple strips in each iteration (an example with double buffering is shown in Figure 1). In this example the kernels exhibit producer-consumer locality producing intermediate streams. As shown in Figure 1, all operations on the intermediate streams are contained within each iteration of the control thread loop. Hence, iterations are independent and could be scheduled on different LMs.

Figure 4 illustrates how the stream program of Figure 1 is decomposed at runtime into work queue entries and dependencies. In this example two Kernel Processors share an LM. The control thread has split stream data into n strips and enqueued tasks and dependencies on the memory and compute work queues. Each kernel invocation operates on one half of a strip (e.g., sub-strips $\mathbf{K1_{0,0}}$, $\mathbf{K1_{0,1}}$) to spread work across the two Kernel Processors. Each work queue entry contains the arguments to the matching calls in Table 1 along with dependency bitmaps. For example, dependency information for kernel $\mathbf{K1_{0,0}}$ indicates that it cannot execute until loads/gathers to its input strips ($\mathbf{La_0}$, $\mathbf{Li_0}$ (index strip), $\mathbf{Gc_0}$) are completed.

There are several options for spreading work across the LMs. For example, consecutive control loop iterations could be interleaved across the set of LMs or randomly distributed across the LMs. These approaches have the disadvantage that when kernels need to access data from adjacent strips (e.g., for stencils or convolution operations), the data are located in a different LM resulting in LM cache misses. An alternative approach is to divide the control thread loop iterations into groups of consecutive iterations, where the groups are mapped one-to-one to the set of LMs. This approach minimizes LM cache misses due to kernels accessing adjacent strips. However, it has the limitation that work is statically assigned to each LM, leading to poor performance for variable rate kernels and other workload variations.

For Streamware, the approach we take is to use piecewise splitting in which limited size groups of consecutive control thread loop iterations are assigned dynamically to LMs. The consecutive iterations in each group place operations together for the same LM filling its work queues until it cannot accommodate the next iteration. After the control thread fills the work queues, it calls `executeWork`. Kernel and memory operations execute until the work enqueued for any LM is completed, causing `executeWork` in the control thread to return the ID of the completed LMs. The control thread then refills the completed LM work queues with operations in the next group of consecutive control thread loop iterations. This approach provides dynamic scheduling which adapts to different rates of execution by processor cores and DMA engines (e.g., because of variable rate kernels and other workload variations). Maximally filling the work queues each time they empty minimizes contention for service by the control thread. Assigning groups of consecutive iterations enables kernels to access adjacent strip data from the local LM without incurring cache misses except at the boundary between groups, and provides efficient support for associative reductions.

The runtime system design can be extended to support both stateful kernels and stateless kernels. In general, a stateful kernel can be handled by using space multiplexing in which threads are dedicated to the producer and consumer kernels which process entire streams before relinquishing the threads (Section 2). Equivalent behavior can also be accommodated in a time-multiplexing approach like ours by scheduling the stateful producer-consumer kernels to run in repeated invocations that operate on strips. State is carried across invocations using persistent static variables. Whereas different control thread loop iterations are independent for stateless producers-consumers and can therefore be scheduled using different LMs, stateful producers-consumers have dependencies. As a result, the runtime must be extended either to restrict the stateful kernels to execute using the same LM, or extended to support communication between LMs (which translates to cache-to-cache communication in our runtime mapping). A special case of stateful kernels is reduction operations. We implement reduction in two phases. In the first phase each core performs reduction of all the kernel invocations it executes. The reduction data is collected in the second phase using simple associative combination or using tree combining (also associative).

4. Evaluation

In this section we present a performance evaluation of the Streamware runtime system on multicore general-purpose processors. Overall, the results show that for most cache configurations, the runtime system provides good application scalability with increasing numbers of cores for compute-intensive applications, with less improvement for memory-intensive workloads. The runtime scheduler efficiently distributes work across cores, and the compute resources within each core are used effectively by exploiting short-vector SIMD units. We also evaluate the performance overheads due to the runtime system, consider the effects of scaling stream local memories within caches, and explore dynamic workload balancing among cores with a competing workload.

4.1 Experimental Setup

We evaluate Streamware running on real multicore processor hardware. In addition, we evaluate Streamware using a processor simulator in order to experiment with various configurations of cores and caches.

For experiments on the real system, we use a machine with two processor sockets connected by Intel's 6.4GB/s Front Side Bus, 8GB DDR2 RAM, and running RedHat FC6 (2.6 SMP kernel). Each socket has a dual-core Intel Xeon 5140 (Woodcrest) processor (64-bit, 4MB L2 cache shared by the two cores, SSE3-compatible SIMD units per core). Despite having two dual-core processors,

Table 2. Simulator baseline machine parameters

Parameter	Value	Parameter	Value
Num cores (C)	1,2,4,8,16	Num DMAs	1,2
Core freq.	2GHz	DRAM arch.	DDR2
Pipeline	OOO	DRAM bandwidth	6.4GB/s
L2 size	C*1MB	DRAM burst	32/64 bytes
L2 assoc.	8	DMA AG bandwidth	2addr/cycle
L2 line size	64 bytes	DMA ABs	8 (512 bytes)
FSB bandwidth	6.4GB/s	DMA MSHRs	256

we can emulate a quad-core machine effectively because there is very little cache data sharing in the runtime system and in our test applications.

We spawn one software thread (using Linux pthreads) for each processor core and pin them to separate cores using OS scheduling affinity system calls (thus avoiding high context switching overheads). All threads execute kernels, but some threads are assigned to also execute stream memory operations. SSE packing and alignment is performed as part of stream memory operations. Each strip is automatically quad-word aligned and has length that is a multiple of a quad-word.

One thread is designated the control thread which performs enqueueing operation in addition to executing kernels and memory operations. For efficient thread coordination one spin-lock per LM is used to protect access to the LM kernel work queue, and atomic instructions provided by the x86 ISA are used to update bits in dependency vectors. No other locking is needed in the implementation, and in fact an alternative wait-free implementation of the work queues could be implemented. The dependency vector is 64 bits limiting each work queue to 64 entries.

For simulation experiments, we enhanced the M5 simulator [8] to support various architectural configurations (scalable number of cores and caches) and to include a special Stream Load-Store (SLS) hardware unit for stream memory operations. M5 is a cycle-accurate system-emulation/full-system simulator of multicore general-purpose processors and was configured to execute the Alpha instruction set in system-emulation mode. We augmented M5 by integrating the DRAMsim DRAM simulator [39] to accurately simulate DRAM throughput since memory bandwidth is a critical metric when evaluating stream programs. The baseline machine parameters used for our simulations, including number of processor cores and number of DMAs, are detailed in Table 2. In addition, the simulation may be a closer match for future hardware including SLS units and hardware multithreading (SMT) which are attractive because they can provide ideal overlap of memory and kernel operations.

We evaluate four scientific applications that feature regular and irregular mesh constructs and linear algebra operations. The applications were originally written by programmers in the areas of fluid dynamics and solid mechanics. The four applications and characteristics of the datasets are summarized in Table 3, and further details are available in [18]. These applications are significantly challenging because they contain non-affine and data-dependent array references, along with a wide range of compute to memory ratios. FEM and NEO are relatively compute-intensive applications and CDP and SPAS are more memory-intensive applications.

The overall evaluation methodology is as follows. The C/Fortran conventional implementations are first re-written in a stream-programming style [11]. We transform the stream program using standard stream compiler transformations as shown in Figure 2 into stream virtual machine code similar to Figure 1. Both the conventional and stream codes are compiled using the Intel C Compiler for Linux (icc) with -O3 level optimizations to generate

Table 3. Application and dataset description

FEM [6]: 2D Discontinuous Galerkin finite element method code for fluid dynamics. It includes both a 4 816 and 80 001 element unstructured mesh, and solves for either the *Euler* or *magnetohydrodynamics* (MHD) equations. The code is parametrized for linear (lin), quadratic (qd), or cubic (cub) interpolation. FEM performs mostly gather and scatter memory operations of records spanning $5-80$ words and has three compute-intensive kernels.
CDP [23]: 3D *large eddy* fluid dynamic finite volume method simulation on an irregular mesh. The *finer_nb* and *pw_6000_nb* datasets have a mix of tetrahedrons, prisms, pyramids, and cubic elements with 3 800, 29 095, and 1 278 373 total control volumes. CDP performs gather, scatter, and scatter-add memory operations to records of $1-8$ words and has four kernels.
SPAS [38]: Part of a sparse algebra suite; computes a compressed sparse row matrix vector multiplication on a 9 978, 19 094, 37 918, or 73 053, 435 382 row matrix. SPAS uses unit-stride stream loads and stores and has one main kernel.
NEO [7]: A *neo-hookean* solid mechanics code that models a finite elasticity compressible material. The application uses a structured grid with 30 000, 50 000, 100 000, or 200 000 elements. NEO uses unit-stride stream loads and stores and has five kernels.

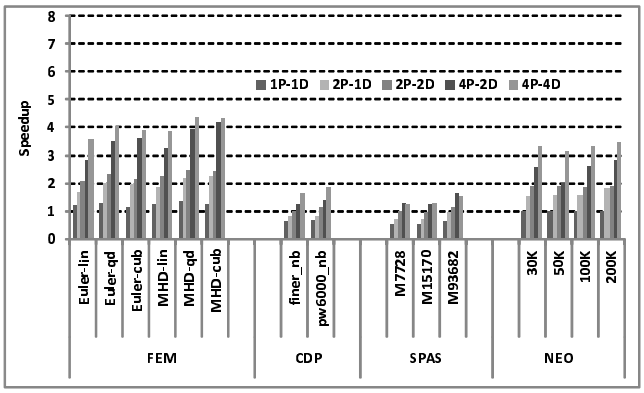

Figure 5. Streamware performance on the Xeon processor

optimized x86 binaries. For the simulator, we run alpha-gcc 2.95.3 with -O3 level optimizations to generate optimized alpha binaries. These binaries are then run to collect execution statistics.

4.2 Evaluating Streamware on Xeon

In this section we demonstrate the portability and performance gains of Streamware on the 2-socket Xeon system. We separate the benefits obtained by auto-parallelization and dynamic scheduling across cores from the additional benefits obtained by efficiently exploiting the SSE units of the individual cores.

4.2.1 Stream Execution

Streamware should give good performance while providing portability across underlying hardware architectures. To evaluate these benefits we run the same applications for several processor-DMA configurations with various data sets on our two-socket Xeon machine. Figure 5 presents the speedups of the stream version of each application over the conventional single-threaded C version for 1P-1D (one processor, one DMA), 2P-1D, 2P-2D, 4P-2D, and 4P-4D configurations. These configurations do not necessarily correspond to physical analogs on our machine, but they map to the underlying hardware to provide different levels of core and bandwidth utilization. For example, 4P-4D represents four logical processors and four logical DMA units where each DMA unit feeds a local memory emulated by a portion of L2 cache, effectively using two DMAs per L2 cache. Configurations with two DMAs (2P-2D and

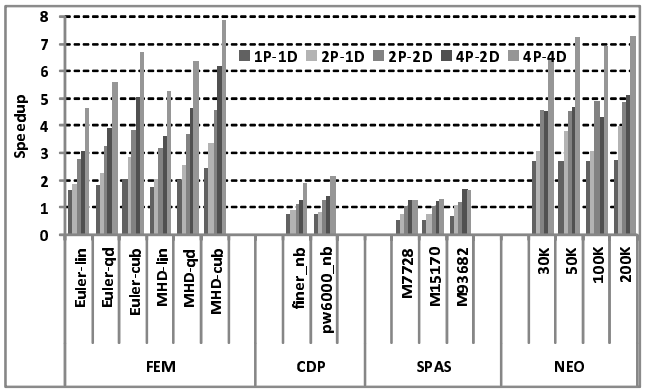

Figure 6. Streamware performance with SSE on the Xeon processor

4P-2D) allocate one DMA per socket by pinning the corresponding logical processors to the physical processors in that socket.

Across applications, speedups are highest for the large data sets since they typically do not fit in the L2 caches and therefore show the benefits of staging data in local memories. Small data sets which do fit in the cache do not perform as well because conventional codes perform well in these cases.

Our two compute-intensive applications, FEM and NEO, achieve roughly linear speedup (4.3x and 3.5x, respectively) for the 4P-4D configuration. The performance of FEM and NEO is limited by the compute bandwidth (FP throughput) of the processor cores. An interesting point is that we see greater than linear speedup (i.e. 1P-1D runs 1.1x to 1.4x faster) for the computationally demanding FEM data sets because the stream programming model more efficiently supports producer-consumer locality in these applications compared to the conventional code.

CDP and SPAS, both memory-intensive applications, achieve only 1.9x and 1.55x speedup for the 4P-4D configuration as they are limited by the memory system bandwidth. For 1P-1D, the performance of CDP and SPAS is actually worse (speedup < 1) than that of the conventional codes. Performance is limited in this single-processor configuration by the inability to overlap the execution of compute and memory bulk operations, and these applications have little producer-consumer locality to exploit. CDP and SPAS also have worse performance than the conventional code in the 2P-1D configuration, but speedups are present when there are more DMA threads.

For all the applications, performance increases with larger numbers of DMA threads until the memory bandwidth is saturated. Although we had expected that configurations with a single DMA thread per L2 cache would be able to saturate the memory bandwidth, we found that often multiple DMA threads per cache were needed (configurations 2P-2D and 4P-4D use two DMA threads per cache). The performance improvement of 2P-2D over 2P-1D is due not only to the presence of an additional DMA thread, but also to our implementation of 2P-2D which mapped each processor/DMA to a separate socket, increasing LM size and memory bandwidth. In Section 4.3.1, we run our experiments with a hardware DMA unit in the M5 simulator to saturate the memory bandwidth and avoid this effect.

4.2.2 Exploiting Short-Vector SIMD Units

In addition to enabling efficient execution across processor cores, the stream programming model coupled with Streamware enables better utilization of each core by exploiting the short-vector SIMD units (e.g., SSE units in the x86 processors). Using the stream programming model greatly simplifies automatic compiler generation of SSE instructions by removing non-affine accesses within kernels. Streamware further facilitates automatic SIMDization by performing data packing and re-alignment on bulk memory operations to match the SIMD boundaries (Sec 3.2). We evaluate these effects by manually emulating a compiler pass with simple transformations that convert instructions inside kernels to SSE intrinsics and show the potential performance benefits.

Figure 6 shows the performance gain for stream programs with SSE intrinsics inside kernels over the conventional codes. As expected, compute-intensive workloads benefit greatly from SSE instructions while memory-intensive workloads show much less improvement. For the 4P-4D configuration, FEM and NEO achieve speedups of 7.8x and 7.3x respectively for their most challenging data sets. Euler and MHD linear data sets for FEM show speedup around 5x due to less computational intensity. CDP and SPAS exhibit almost no additional speedup for using SSE intrinsics because they contain data-dependent computations (i.e. an unknown number of non-zero elements per row of the input matrix in SPAS and a variable number of neighbors per element in CDP) which make poor use of the SIMD units.

The results in Figure 6 do not achieve the theoretical maximum of 4x speedup over non-SSE code. Obviously the memory-intensive applications cannot achieve 4x speedup. However, compute-intensive applications have sub-optimal speedup because using SSE increases the computational bandwidth of the cores, putting more pressure on the memory bandwidth. This is made evident by the practically non-existent performance gains of NEO when going from 2P-2D to 4P-2D. FEM does not suffer in the same way because it is computationally more intense. Overall, effective memory bandwidth is reduced when we use SSE because packing is performed as data is brought into the local memories. Since data is not accessed sequentially during packing, we cannot take advantage of bulk `memcpy` operations and the hardware prefetcher. This effect is significant when we do not have enough DMA threads to saturate the memory bandwidth.

4.3 Analysis of Key Design Issues

In this section we stress-test our Streamware system to analyze the major design issues. We evaluate how Streamware performs as the number of processor cores/caches increases, the main sources of overheads, and how Streamware adapts to the presence of competing workloads.

4.3.1 Scaling with Processor Cores

A key figure of merit for our runtime system is how well it scales with multicore processor configurations. As the number of cores and caches varies, different aspects of the application become the performance bottleneck. Figure 7 shows speedup with a varying number of processor cores (P) and DMAs (D) for the FEM and SPAS applications. We chose these two applications to illustrate the effect of both compute-intensive and memory-intensive workloads. Using the M5 simulator, we scaled the number of cores from one to 16 because we believe this range represents multicore architectures that will be available in the immediate to near future. Each DMA corresponds to a local memory serving a set of processor cores. While a single hardware DMA unit can saturate the memory bandwidth, we simulated up to two DMAs to reduce contention for the lock when many processors operate from a single work queue. We also scaled the LM size linearly with the number of cores.

The performance of FEM scales fairly well with an increasing number of cores and DMAs (greater than 12x speedup for 16 cores with two DMAs) because the application is compute-intensive. SPAS, on the other hand, is memory-intensive and therefore shows much lower performance gains, achieving only 4x speedup for 16

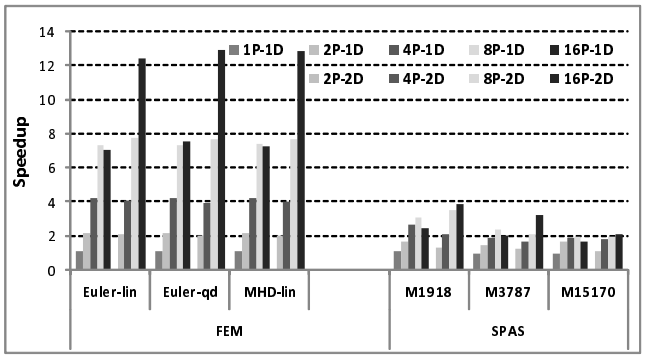

Figure 7. Scaling of Streamware with processor cores/DMAs

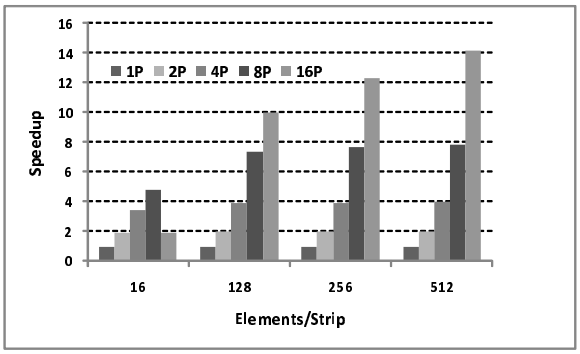

Figure 8. Overhead of Streamware runtime

cores with two DMAs. Overall, performance with two DMAs is better than that with a single DMA for large number of cores ($>=$ 8). This is because the cores connected to the same DMA share a common kernel work queue and contend with each other for work. When the ratio of cores to DMAs is large, most cores spend time blocked until they can execute. This is evident in the performance of all three FEM data sets for the 8P and 16P configurations. Due to the memory-intensive nature of SPAS, performance does not increase as much with the increase in number of DMAs.

4.3.2 Overhead of Streamware Runtime System

In Figure 8, we evaluate the overhead of the Streamware runtime scheduler. For this experiment, we created a synthetic benchmark which simulates the concurrent execution of kernels on multiple cores. A kernel cannot execute until it obtains the lock for the queue and checks a bit vector to verify that its dependencies have been satisfied. When the kernel is finished, it clears its dependencies in the bit vectors. The results presented in Figure 8 reflect the total overhead of the runtime system (overhead due to locks, handling dependencies, etc). We found that contention for the queue lock is the dominant source of the overhead.

Lock overhead is relative to the amount of work per kernel operation, which we represented as 16, 128, 256, and 512 elements/strip with 400 cycles of kernel computation per element. The number of elements per strip is divided evenly between the processors in each run of the experiment. Figure 8 shows the speedup of kernel execution on multiple cores. With 16 cores, contention for the lock is highest and we see sub-optimal speedup for low amounts of work. The pathological case is 16 elements/strip and 16 cores, where we operate on only a single element per core and see 2x speedup. A scenario more representative of our workloads is 512 elements/strip which shows a speedup of 14x. Compared to the theoretical maximum speedup of 16x, this overhead seems acceptable. Since locks do not scale well for more than a few cores, we can ameliorate lock contention by increasing the number of local memories (one lock per local memory) while keeping the total local memory size constant, as shown in the previous section. However, this comes at the cost of decreasing the number of elements/strip which could negatively impact performance due to short stream effects (e.g., more kernel invocations).

4.3.3 Dynamic Workload Balancing

Up to this point, all of our experiments have been performed with only a single workload on the system, but a workload executing in the presence of competing jobs is a more realistic scenario that shows the benefits of our dynamic scheduling approach. So, we run a synthetic stream benchmark concurrently with an additional process on a 4P-4D configuration on Xeon. This ensures that part of the stream benchmark execution is preempted by the OS because

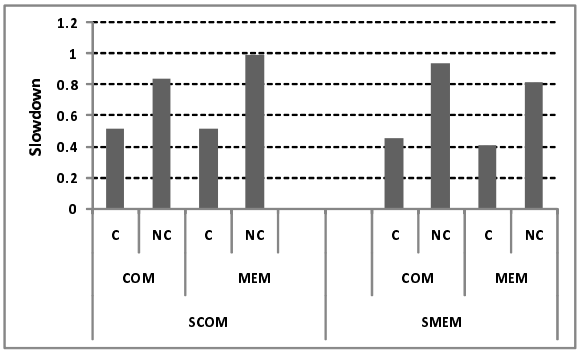

Figure 9. Dynamic workload effects on Streamware

the stream benchmark uses all 4 cores of Xeon. This experiment is, in some sense, a generalization of having heterogeneous cores where specific cores are consistently slower/faster.

For both the stream benchmark and the competing process, we vary the compute to memory ratio to illustrate the effects of shared hardware resources. We also run the experiment for two placements of the competing process: (i) on the same processor as the control thread (C), and (ii) on a processor not containing the control thread (NC).

Figure 9 shows the performance slow down for a compute-intensive stream program (SCOM) and a memory-intensive stream program (SMEM) in the presence of a competing process that is either compute (COM) or memory (MEM) intensive. As one would expect, performance is worst ($0.4 - 0.5$x) when the competing process runs on the same processor core as the control thread of the stream benchmark. This blocks other processors from doing work beyond what is currently scheduled in the queue. Conversely, the best performance ($0.9 - 1.0$x) is seen when the two processes are complementary, namely one is COM and the other is SMEM, and vice versa (i.e. one is MEM and the other is SCOM). When the stream benchmark is memory-intensive, a competing process that is also memory-intensive halts one processor from doing useful work and pollutes the local memory (L2 cache) to degrade performance to 0.8x. There is less cache pollution when the SCOM benchmark is run with the COM process. An option to mitigate the effects of a concurrent process on stream application performance is to elevate the scheduling priority of the control thread in the OS.

5. Discussion and Related Work

Much earlier work in scheduling stream programs has been confined to the context of stream processors [20, 13, 27, 3, 36], where stream compilers [16, 14, 37, 26, 29, 32] make scheduling deci-

sions and specialized hardware (e.g., MFCs in Cell, Scoreboards in Imagine/Merrimac) handles the dispatch of bulk compute/memory operations. In contrast, Streamware is a software run-time targeting general-purpose processors. Stream scheduling is mostly performed at run-time and is informed by the specific properties of the platform hardware (e.g., number of processors per LM, sizes of LMs). This approach enables Streamware to adapt to dynamically changing workloads and also makes it portable to diverse multicore processor configurations. Some recent research efforts have proposed methods to run stream programs on general-purpose uniprocessors with SMT-capable hardware or specialized hardware extensions (e.g., SLS units) [18, 17]. Our work builds on these efforts to extend to multicore processor platforms that can differ in processor/cache memory configurations.

With the increase in the on-chip parallel substrates, using runtime systems to automatically map to different multicore configurations is becoming increasingly popular [10, 35, 12]. These approaches mainly focus on providing support for multi-threading programming models. For example, McRT [10] provides a runtime system based on user-level thread libraries and task queues to schedule work on multicore processors. Thread clustering [35] focuses on using software/hardware performance counters to colocate threads sharing the same cache. Phoenix [12] exclusively focuses on a runtime for Google MapReduce using mutually independent threads. The very recently and independently proposed MSL framework in [40] uses several techniques that are similar to Streamware, but currently targets Cell processor and presents preliminary performance results. Cilk [15] provides extensions to C for programmers to express parallelism, which is then automatically parallelized by the scheduling runtime. Although we share a similar high-level vision as these approaches, Streamware exclusively focuses on the stream programming model for general-purpose processors and effectively uses the exposed parallelism and locality inherent to the model to distribute work into memory/compute work queues and execute tasks in parallel.

The idea of using runtime systems to map to a variety of multiprocessors/clusters has been extensively studied. OpenMP [4]- and MPI [2]- based runtime systems have traditionally been used in HPC for large systems. More recently, several languages and runtime systems (e.g., Berkeley Titanium [24], Microsoft Dryad [31], IBM X10 [33], Sequoia [30], Sun Fortress, Cray Cascade) are being developed to provide portability and programmability. These efforts are mainly focused on multi-node configurations where each node could be a multi-core processor. While Streamware uses some similar techniques (e.g., for load balancing) it is distinguished by focusing on optimizing single node performance by effectively utilizing the on-chip caches and SIMD execution units of the multicore processors using the stream programming model.

Traditionally multithreading (e.g., using pthreads) has been used to partition a program into multiple units of work. However, as the number of threads increases it becomes increasingly difficult to handle synchronization and communication between threads, especially using fine grained locks. Recently, multi-threading programming models such as Transactional Memory [9, 19] and Intel Thread Building Blocks [1] have been proposed to ease concurrent programming challenges. Streamware also sidesteps most complexity associated with locks because they are needed only at the coarse granularity of large computation (kernels) blocks or large memory blocks (DMAs); moreover, these low-level locks are in the runtime system and not exposed to the programmer. Finally, since stream programming explicitly exposes the computation and memory accesses, Streamware effectively overlaps computation with memory accesses.

6. Conclusions

With the onset of the multicore era, as the number of on-chip resources increases, it becomes increasingly important to provide a way to efficiently utilize them. With hardware vendors introducing diverse architectures for next generation processors, it is also important that applications are portable across different hardwares.

In this paper, we presented Streamware, a software system utilizing the stream programming model to efficiently execute applications on multi-core architectures. We defined a low-level stream virtual machine API which can either be targeted by a high-level stream compiler or can be directly used to write applications in the stream style. We use cache hierarchy aware, dynamic scheduling to automatically parallelize and efficiently map applications to the hardware based on its configuration and the current workload.

Using both real hardware and a cycle-accurate simulation system, we demonstrated that Streamware enables compute-limited applications to achieve nearly linear speedup with an increasing number of cores, and also improves the performance of memory-intensive applications limited only by the memory bandwidths offered by these processors. Our work demonstrates that the stream programming model, coupled with an efficient runtime system, is an excellent alternative to conventional models for current and emerging multicore processors.

7. Acknowledgements

We would like to thank Timothy Barth of NASA Ames and Eric Darve of the Mechanical Engineering Department at Stanford University for their valuable help in providing applications and working with us for their stream implementations. Many thanks to our colleagues in Stanford's Merrimac project for their useful feedback throughout this work. Thanks also goes to the reviewers for their insightful comments which significantly helped in improving the quality of the presentation. This work was supported, in part, by the US Department of Energy ASC Alliances Program, contract LLNL B523583, with Stanford University.

References

[1] Intel Thread Building Blocks. osstbb.intel.com.

[2] MPI. www.open-mpi.org.

[3] NVidia G80. www.nvidia.com.

[4] OpenMP. www.openmp.org.

[5] RStream Compiler. www.reservoir.com.

[6] T. Barth. Simplified discontinuous Galerkin methods for systems of conservation laws with convex extension. In *Discontinuous Galerkin Methods*, volume 11 of *Lecture Notes in Computational Science and Engineering*. Springer-Verlag, Heidelberg, 1999.

[7] Y. Basar and M. Itskov. Constitutive model and finite element formulation for large strain elasto-plastic analysis of shells. In *Journal of Computational Mechanics*, Jun 1999.

[8] N. Binkert, E. Hallnor, and S. Reinhardt. Network-oriented full system simulation using M5. In *CAECW*, 2003.

[9] Bratin Saha et al. McRT-STM: a high performance software transactional memory system for a multi-core runtime. In *PPoPP*, 2006.

[10] Bratin Saha et al. Enabling scalability and performance in a large scale CMP environment. In *Eurosys*, 2007.

[11] I. Buck, T. Foley, D. Horn, J. Sugerman, K. Fatahalian, M. Houston, and P. Hanrahan. Brook for GPUs: Stream computing on graphics hardware. In *SIGGRAPH*, 2004.

[12] C. Ranger et al. Evaluating MapReduce for Multicore and Multiprocessor Systems. In *HPCA*, 2007.

[13] W. Dally, P. Hanrahan, M. Erez, T. J. Knight, F. Labonte, J.-H. Ahn, N. Jayasena, U. J. Kapasi, A. Das, J. Gummaraju, and I. Buck. Merrimac: Supercomputing with streams. In *SC*, Nov 2003.

[14] A. Das, W. Dally, and P. Mattson. Compiling for Stream Processing. In *PACT*, 2006.

[15] M. Frigo, C. E. Leiserson, and K. H. Randall. The implementation of the Cilk-5 multithreaded language. In *PLDI*, 1998.

[16] M. Gordon, W. Thies, and S. Amarasinghe. Exploiting coarse-grained task, data, and pipeline parallelism in stream programs. In *ASPLOS*, 2006.

[17] J. Gummaraju, M. Erez, J. Coburn, M. Rosenblum, and W. Dally. Architectural Support for the Stream Execution Model on General-Purpose Processors. In *PACT*, 2007.

[18] J. Gummaraju and M. Rosenblum. Stream Programming on General-Purpose Processors. In *International Symposium on Microarchitecture*, 2005.

[19] M. Herlihy and J. E. B. Moss. Transactional memory: Architectural support for lock-free data structures. In *ISCA*, 1993.

[20] H. P. Hofstee. Power efficient processor architecture and the Cell processor. In *HPCA*, Feb 2005.

[21] J. Leverich et al. Comparing Memory Systems for Chip Multiprocessors. In *ISCA*, 2007.

[22] K. Fatahalian et al. Sequoia: Programming the Memory Hierarchy. In *SC*, Nov 2006.

[23] K. Mahesh et al. Large eddy simulation of reacting turbulent flows in complex geometries. *ASME J. of Applied Mechanics*, May 2006.

[24] K. Yelick et al. Titanium: A high-performance Java dialect. In *ACM Workshop on Java for High-Performance Network Computing*, Feb 1998.

[25] U. Kapasi, W. Dally, S. Rixner, J. Owens, and B. Khailany. The Imagine stream processor. In *ICCD*, Sep 2002.

[26] F. Labonte, P. Mattson, I. Buck, C. Kozyrakis, and M. Horowitz. The Stream Virtual Machine. In *PACT*, 2004.

[27] M. B. Taylor et al. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs. *IEEE Micro*, 22:25–35, March 2002.

[28] M. Erez and J. Ahn and J. Gummaraju and M. Rosenblum and W. Dally. Executing Irregular Scientific Applications on Stream Architectures. In *ICS*, 2007.

[29] M. Gordon et al. A Stream Compiler for Communication-Exposed Architectures. In *ASPLOS*, 2002.

[30] M. Houston et al. A Portable Run-time Interface for Multi-level Memory Hierarchies. In *PPoPP*, 2008.

[31] M. Isard et al. Dryad: Distributed Data Parallel Programs from Sequential Building Blocks. In *Eurosys*, 2007.

[32] M. D. McCool. Data-parallel programming on Cell BE and the GPU using the Rapidmind development platform. In *GSPx Multicore Applications Conference*, 2006.

[33] P. Charles et al. X10: An object-oriented approach to non-uniform cluster computing. In *OOPSLA*, 2005.

[34] T. Knight et al. Sequoia: Programming the Memory Hierarchy. In *PPoPP*, 2007.

[35] D. Tam, R. Azimi, and M. Stumm. Thread Clustering: A Share-aware Scheduling on SMP-CMP-SMT Multiprocessors. In *EuroSys*, 2007.

[36] D. Tarditi, S. Puri, and J. Oglesby. ACCELERATOR: Using data-parallelism to program GPUs for general-purpose uses. In *ASPLOS*, 2006.

[37] W. Thies, M. Karczmarek, and S. Amarasinghe. StreamIt: A language for streaming applications. In *ICCC*, 2002.

[38] R. Vuduc, J. W. Demmel, K. A. Yelick, S. Kamil, R. Nishtala, and B. Lee. Performance optimizations and bounds for sparse matrix-vector multiply. *SC*, 2002.

[39] D. Wang, B. Ganesh, N. T. K. B. A. Jaleel, and B. Jacob. DRAMsim: A memory system simulator. In *SIGARCH Computer Architecture News*, September 2005.

[40] D. Zhang, Q. Li, R. Rabbah, and S. Amarasinghe. A Lightweight Streaming Layer for Multicore Execution. In *Workshop on Design, Architecture, and Simulation of Chip Multiprocessors*, Dec 2007.