Enabling Scalability and Performance in a Large Scale CMP Environment

Bratin Saha*, Ali-Reza Adl-Tabatabai*, Anwar Ghuloum*, Mohan Rajagopalan*, Richard L. Hudson*, Leaf Petersen*, Vijay Menon*, Brian Murphy*, Tatiana Shpeisman*, Eric Sprangle+, Anwar Rohillah+, Doug Carmean+, Jesse Fang*

*Programming Systems Lab, +Digital Enterprise Group
Intel Corporation

{firstname.lastname} @intel.com

Abstract

Hardware trends suggest that large-scale CMP architectures, with tens to hundreds of processing cores on a single piece of silicon, are iminent within the next decade. While existing CMP machines have traditionally been handled in the same way as SMPs, this magnitude of parallelism introduces several fundamental challenges at the architectural level and this, in turn, translates to novel challenges in the design of the software stack for these platforms. This paper presents the "Many Core Run Time" (McRT), a software prototype of an integrated language runtime that was designed to explore configurations of the software stack for enabling performance and scalability on large scale CMP platforms. This paper presents the architecture of McRT and discusses our experiences with the system, including experimental evaluation that lead to several interesting, non-intuitive findings, providing key insights about the structure of the system stack at this scale. A key contribution of this paper is to demonstrate how McRT enables near linear improvements in performance and scalability for desktop workloads such as the popular XviD encoder and a set of RMS (recognition, mining, and synthesis) applications. Another key contribution of this work is its use of McRT to explore non-traditional system configurations such as a light-weight executive in which McRT runs on "bare metal" and replaces the traditional OS. Such configurations are becoming an increasingly attractive alternative to leverage heterogeneous computing uints as seen in today's CPU-GPU configurations.

Categories & Subject Descriptors: D.1.3:Programming Languages: Parallel Programming, D.3.4 Processors: Run-time environments, D.4.0 Operating Systems: General

General Terms: Measurement, Performance, Design, Experimentation

Keywords: Multi-core processors, runtime design, scheduler design, parallel programming, synchronization primitives, transactional memory, memory management, sequestered mode

Eurosys'07, March 21-23, 2007, Lisbon, Portugal.

1. Introduction

Hardware architects are increasingly turning to CMP based solutions as single threaded processor performance becomes power limited [32]. Apart from power, computing requirements of emerging domains such as gaming, multi-media and data mining motivate the need for exploring multi-threaded CMP-based solutions [32]. The *tera-scale computing* research project at Intel [33] explores futuristic processor configurations and will potentially deliver CMP platforms with 10s to 100s of processing units within the next decade. The first step in this transition, which we call "small scale CMP", involves integrating multiple conventional cores on a single chip. Several such products have already been announced and are expected to enter the markets by the end of the year. Small-scale CMPs aim to boost performance without stretching the power envelope.

While improving power efficiency, small-scale CMPs still can not fulfill the computing needs to enable new usage models based on multimedia, games, data mining, etc. [32]. This motivates the need for "large-scale" CMPs, also called many-core platforms, that integrate tens of multi-threaded cores on a single chip. Unlike conventional processors, each core in a large-scale CMP may potentially sacrifice single-threaded performance to improve overall efficiency. This new architectural paradigm introduces several new challenges in the design of the software stack for large-scale CMPs. For example, these architectures may remove out-of-order mechanisms yet achieve high throughput by threading the individual cores.

This paper presents the "Many-core Run Time" (McRT, pronounced '*mac-ar-tee'*), a software prototype of an integrated language runtime that was designed to explore configurations of the software stack for enabling performance and scalability on large scale CMP platforms. A key realization from this work was that enabling scalability and performance required a high degree of synergy between the different components that make up the system. This paper presents the architecture of McRT and discusses our experiences with the system, including experimental evaluation that lead to several interesting, non-intuitive findings. The primary contributions of this work are:

1. McRT is a new system software framework that is designed to study the performance characteristics on large scale CMP platforms. The novelty of the framework lies in the synergy between the different components, rather than in the components themselves. We had to significantly re-architect the system software stack to allow for a tight integration between the different components of McRT.
2. We show that McRT required non-intuitive mechanisms for enabling scalability on threaded workloads; for example, McRT needed to implement different runtime policies depending on the number of underlying hardware threads.
3. A key distinguishing aspect of this work has been its focus on emerging desktop application workloads rather than on traditional throughput oriented task-parallel systems workloads such as web-servers and databases. (In the rest of this paper, we will refer to the emerging desktop workloads as RMS (recognition, mining, synthesis) workloads [41]).
4. Finally, we present the design and results for "*sequestered execution mode"*, where McRT functions as a light-weight runtime system that runs directly on "bare metal" hardware without an underlying OS. In this mode, the application can still access OS services such as file I/O, while leveraging the lightweight nature of McRT to enable better scalability. Preliminary results with sequestered mode argue for re-evaluating both the role and structure of operating systems on large scale CMP platforms.

1.1 Motivation

While system software has traditionally treated CMPs as "SMPs on chips" [5][6], there are fundamental differences in the characteristics of these platforms. Our experience shows that it is critical to address these differences in order to implement a scalable and effective software stack. In particular, the software stack needs to address three factors: support for fine-grained parallelism, programmability enhancements, and different kinds of heterogeneity.

Fine-grained parallelism:

Architectural differences make it essential to provide efficient fine-grained thread interactions on large scale CMP systems. For example, the compute-to-cache ratio (number of HW threads / aggregate cache size) for a large scale CMP platform is orders of magnitude higher than for traditional SMPs. The latency to access data from a different HW thread is an order of magnitude lower than a traditional SMP. Finally, the instruction issue bandwidth has a high premium in many-core processors since, unlike traditional SMPs, each processor is highly threaded. Section 3.2 explains these differences in greater detail.

In addition, the usage model for emerging application domains also motivates the need for fine-grain parallelism. While these applications do not benefit from traditional coarse grained task-parallel techniques [34], they benefit greatly from fine-grained models such as nested data parallelism.

Programmability enhancements: Due to the relatively high cost for multi socket systems traditional SMP systems were targeted at niche markets, and ran specific workloads written by sophisticated programmers. In contrast, large scale CMP platforms potentially target main stream price points, and therefore a key challenge is to help mainstream programmers embrace parallelism. This motivates the need to include features that help average programmers deal with the challenges of parallel programming.

Heterogeneity: Large scale CMP platforms are likely to run a more diverse set of applications than SMP systems simply because they can target a larger, broader market. This has two implications: (a) the software stack can not be customized for individual applications, but must support a heterogeneous set of applications (b) the underlying HW may itself be heterogeneous [39]; for example, it may include a few heavyweight cores with good scalar performance, and many light-weight threaded cores with good compute density (e.g. for RMS applications).

Given the above differences, we believe that the system software stack requires a significant, holistic, redesign in order to exploit the capabilities of large scale CMP platforms. Accordingly, the McRT framework was designed to provide a flexible and configurable framework that can efficiently scale to more than 64 HW threads. McRT also provides a rich set of abstractions that can be used to realize different kinds of execution semantics, including those seen in traditional thread-based or event-based programs. A key feature of the design has been the tight coupling with programming language interfaces – McRT provides a basic set of primitives that can be used to support different programming models such as OpenMP, pthreads, and Java. Finally, McRT addresses the key large scale CMP requirements in the following ways (elaborated in the rest of the paper):

(1) **Fine-grained parallelism**: Internally, McRT implements a significant fraction of the traditional OS functionality at the user-level. For example, it includes a user-level threads package based on co-operative scheduling, a user-level scheduler that supports configurable scheduling policies, and a user-level synchronization framework. In addition, it provides very lightweight primitives, such as futures, that can support efficient fine-grained parallelism constructs at the language level.

(2) **Programmability enhancements**: McRT includes an optimized software transactional memory module that is

integrated with a compiler to support language level transactions. This allows applications to use transactions in lieu of locks. Transactions enhance programmability by eliminating deadlock, automatically providing scalability features such as fine-grained concurrency and read-sharing, and allowing safe and scalable composition of atomic primitives [35][36][37].

(3) **Heterogeneity**: McRT can be configured to implement different runtime policies to suit different applications. It can also be used to partition a platform into different domains with tasks in each domain scheduled independently. The partitioning can also be leveraged to run McRT in "sequestered mode" (Section 5), where it runs directly on the hardware without an OS underneath, thus allowing a tighter coupling between an application and the underlying HW. In such a mode, the operating system runs on only some of the CMP cores (also referred to as non-sequestered cores) and is used to provide facilities such as I/O to the applications running (on top of McRT) on the sequestered cores.

We chose to build the McRT system from scratch instead of modifying an existing operating system such as Linux, or adding extensions to a micro-kernel operating system[14] [29][15]. McRT was built from scratch for two reasons: first, developing the McRT framework independently ensured that the re-factoring of the system software stack was not limited by the complexities and capabilities of any given operating system. Moreover, it allowed us to explore lightweight configurations that eliminate the OS altogether. Second, the key features of the McRT design are the synergy between the components, and the configurability of the individual components. Achieving this level of synergy and configurability would have required a significant rewrite of the existing components and the APIs**.**

The remainder of this paper is organized as follows. Section 2 gives an overview of McRT highlighting the components that are critical for enabling fine-grained parallelism. Section 3 evaluates McRT performance on a large scale CMP platform. Section 4 discusses the programmability support in the McRT framework. Section 5 uses the sequestered mode execution to highlight McRT's support for heterogeneity. Finally, we discuss related work and conclusions.

2. The Many-Core Runtime

This section discusses the different mechanisms implemented in McRT to enable fine-grained parallelism. Figure 1 presents the general structure of McRT. At its core, McRT consists of five user-level components—a scheduler, a memory manager, a synchronization framework, a transactional memory library and a set of primitives for lightweight threading. McRT's external interfaces are implemented as a set of thin translation layers called *adaptors*. The client adaptor interface can be used to map a variety of higher level programming models and parallel programming implementations to the McRT API. Currently, McRT supports a pthreads, OpenMP and Java virtual machine interface. At the other end of the stack, the *host adaptor* interface enables McRT to run on a variety of platforms including traditional operating systems (Linux and MS Windows) on IA-32 based SMP systems, sequestered systems (Section 5), and large scale CMP simulators. These modules are quite typical in any language runtime.

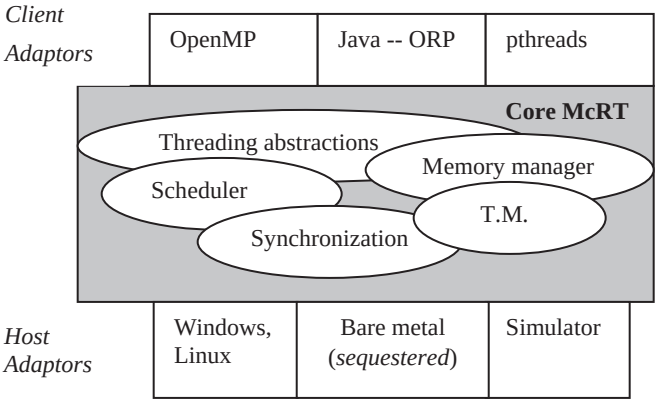

Figure 1: Overview of McRT

McRT provides two basic threading primitives, user-level threads (McrtThread) and futures (McrtFuture), that are tightly integrated with the McRT scheduler. McRT threads provide a fully functional, POSIX compliant implementation of user level threads. For clients that do not require the full threading API and which do not require full co-routine concurrency, an even cheaper form of concurrency is also available through the McRT futures construct. McRT futures are intended to support a common idiom in programming languages, such as CILK[30], in which code is required to have a serial execution, but is also allowed to be run concurrently for performance reasons. The core API provides a "spawn" routine which takes a function and an argument and returns a handle to the result, and a "read" routine which given a handle, returns the result of applying the function to the argument. Because of the serial semantics, McRT has great flexibility as to how it computes the result. It may choose to compute the result eagerly without creating additional parallel work, it may compute the result on demand, or it may compute the result concurrently using additional worker threads. Based on the underlying hardware and system load McRT is able to carefully choose a good scheduling policy that maximizes the granularity of concurrent work while still utilizing as many of the available resources as possible.

McRT provides a highly configurable, user-level scheduler that can be used to realize a variety of scheduling strategies. Section 3 demonstrates that configurability is critical for enabling performance and scalability. As usual the scheduling policy of the system is defined as the mapping between logical processors and task queues. A key aspect of our design has been that the different threading primitives rely on co-operative scheduling instead of the typical pre-emptive scheduler.

Our scheduler framework exports an API to enable a client to dynamically configure the allocation of processing resources to tasks. For example, a client can configure the mapping between logical processors and task queues. Similarly, a client can configure how tasks are added to task queues. Conceptually, three parameters configure the scheduler:

P = number of logical processors,

Q = number of task queues, and

T = number of threading abstractions.

For example, setting P equal to Q, and making processor P_K snoop only the queue Q_k provides processor local task queues. In addition, if a threading abstraction created on a processor P_k is always added to the queue $Q_{k,}$ then the abstractions always *preserve affinity.* On the other hand, allowing a processor P_k to snoop any of the queues provides a *work stealing* scheduler. Adding new threading abstractions in a round-robin way among the task queues provides a *work sharing* scheduling policy.

The task queue abstraction is also used for implementing scheduling domains which can be used to deal with different kinds of heterogeneity. At one level, scheduling domains may be used to tailor the system to HW heterogeneity by using specific domains to represent particular cores with the ability to handle special purpose instructions (e.g. SSE). At a higher level, heterogeneity may be used to indicate different configurations of the software stack. Section 5 shows an example of two distinct kinds of software stacks running on the same system. Formally, to create a domain D_k consisting of logical processors P_i to P_j, a client creates a single task queue Q_k that is snooped only by the processors P_i to P_j. All threading abstractions created by these processors are also added to the queue Q_k. To switch to a different domain D', a threading abstraction uses an API call to yield and enqueue itself to a different task queue (in this case the task queue Q' associated with the domain D'). The abstraction will then get scheduled onto one of the processors belonging to the domain D'. An obvious logical extension is that of clients creating multiple task queues within a scheduling domain.

Another key aspect of the scheduler API is its ability to allow for the fine-grained control and co-ordination between executing tasks. Specifically, the API exports functions that can be used to enqueue tasks on any arbitrary queue, yield control from executing tasks and possibly schedule in new tasks, or change the task-to-queue mappings. Threading abstractions don't get pre-empted; rather the abstractions must yield at intervals to allow other abstractions to execute. ***We chose co-operative scheduling over traditional pre-emptive scheduling for 2 reasons***: first, as shown in Sections 3 & 4, it allows other parts of the McRT framework to use simpler and more efficient algorithms; second, we believe preemption is fundamentally motivated by the need to time-share expensive hardware resources to maximize their utilization. Typically, this motivation would be weakened, often significantly, on a large scale CMP platform due to the large number of HW processors. Co-operative scheduling does come at a cost; for example, it is difficult to guarantee hard real-time constraints, or fairness. The current McRT scheduler does not attempt to provide these properties owing to its focus on performance and scalability, but these are topics of ongoing research.

Our scheduling framework leverages the tight integration of the threading system to provide a richer execution model compared to traditional operating system schedulers. Users can dynamically program scheduler actions – when an abstraction yields, it can pass a predicate to the McRT scheduler to indicate that the predicate must be true before the abstraction is scheduled back. This enables a nice synergy with the synchronization module. The following simplified code sequence from our barrier implementation illustrates this:

```
uint32 barrierWait(Barrier* barrier) {
  threadsLeft =
    lockedDec(barrier->threadsLeftToEnter);
  if (threadsLeft > 1) {
     mcrtYield(barrier->threadsLeftToEnter,
               Equal, 0);
     /* returns when everyone at barrier */
  }
  /* other code */
 }
```

The barrier code first checks whether we are the last thread to enter the barrier (`threadsLeft > 1`). If not, the current thread yields but informs the scheduler to schedule it back only when all threads have reached the barrier. The same mechanism is used by other synchronization primitives, such as, acquiring locks. Yielding through predicates provides a light-weight mechanism for co-scheduling since blocked threads readily yield to active threads.

McRT also includes an elaborate set of synchronization primitives and a high performance memory allocator that replaces native malloc/free implementations. This has been

described in [38]. Each of these components has a tight synergy with the threading system. To enable good caching, the memory manager provides each thread with private allocation blocks. In a large scale CMP system where threads are created in large numbers, these resources need to be recycled. When a thread exits, the scheduler uses a fast callback to notify the memory manager, which adds the blocks owned by the exiting thread to a pool of free blocks (to be recycled as needed).

Finally, Mcrt includes a variety of client side adaptors that can be used to translate programs written in popular paradigms such as pthreads, and OpenMP. The pthreads and OpenMP adapters provide a mostly straightforward translation of the different constructs to the McRT API. The OpenMP adapter supports true nested parallelism. It creates as many McRT threads as specified by any nested parallelism construct, which then get scheduled on the logical processors. The OpenMP adaptor implements the runtime API used by Intel C compiler v8.0. Section 3 describes results for both OpenMP and pthreads programs. A Java Virtual Machine, ORP [40], was also refactored so that the OS threading primitives that it was hitherto using were translated to the corresponding McRT primitives.

3. McRT Performance Evaluation

This section describes experiments that evaluate the performance and scalability of McRT on representative workloads for large scale CMP systems. Since we expect traditional server-side task-parallel workloads such as web-servers to benefit directly from the underlying parallelism, we decided to focus this work on the relatively less-studied client-side workloads, which we believe will drive future many-core architectures. Specifically, our experiments are based on the popular open source MPEG4 encoder XviD (www.xvid.org) and a set of RMS kernels [41][52] for singular value decomposition (SVD) and self organizing maps (SOM). The XviD encoder is used at resolution of 1920x1080 to correspond with frame sizes in the emerging high definition video. We show the performance for encoding the P frames since these (along with the B frames) happen to be the computationally intensive parts of the encoding. SVD has numerous applications in the areas of data-mining and feature extraction, signal processing, automated control; this workload uses the Jacobi method. A SOM is an unsupervised learning method represented by a two-layer neural network. Typically it is used to map N dimensional data to 2 dimensions to discern patterns. It is extensively applied in text and feature mining, pattern recognition and medical diagnostics. The XviD encoder is based on OpenMP, whereas the RMS kernels use pthreads.

3.1 Microbenchmark evaluation

The first set of experiments evaluate the efficiency of McRT's threading primitives through a set of micro-benchmark experiments. Three experiments, also used for evaluating similar user-level threads packages[8], were used: the first experiment measured the cost of thread creation by measuring the cost of creating 255 threads; the second measured the cost of acquiring and releasing locks, computed by performing 1000 consecutive lock and release operations; and finally, the third experiment evaluates the overheads imposed by context switches between threads. This was done by measuring the cost for 1000 context switches between two threads. In each case the *gettimeofday()* system call was used to perform the measurements.

	Native threads on Linux 2.6.9 µsec	McRT µsec
Thread create (255 iterations)	8960	1841
Mutex lock/release (1000 iterations)	82	81
Context switch (1000 iterations)	3600	748

Table 1: Micro-benchmark experiments

Table 1 compares the overheads seen in McRT with those on a native Linux platform. Column 2 reports the measurements seen by using native threads (pthreads) on a dual CPU, dual core, hyper-threaded Xeon 2.8 Ghz machine running RedHat Enterprise Linux 2.6.9-22ELsmp (NPTL 0.60). Column 3 presents the corresponding measurements observed when running McRT on top of Linux on a 2.8Ghz hyper-threaded Xeon machine. As expected, McRT threading primitives are significantly cheaper than their native-threads counterpart.

Our next experiment measures the scalability of our threading primitives. This experiment focused on measuring the cost of thread creation while varying the number of threads. The experimental setup was the same as described earlier. Figure 2 shows the results of this experiment. Note that McRT thread creation costs are not significantly affected even in the presence of thousands of threads; in fact, McRT is much more efficient at high thread counts. The final experiment compares the overhead of McRT threads to the overhead of McRT futures. To do this, we created batches of futures and threads whose executable

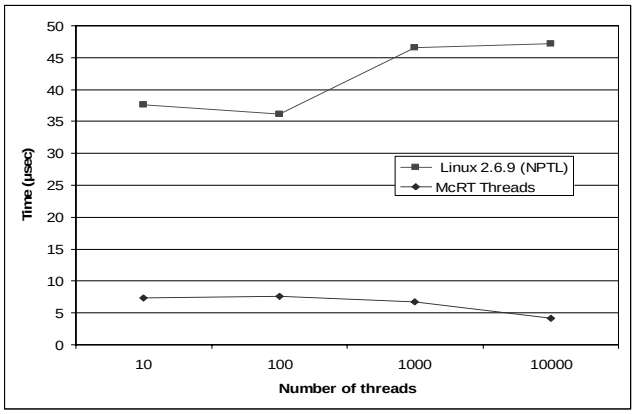

Figure 2: Scalability of McRT threads

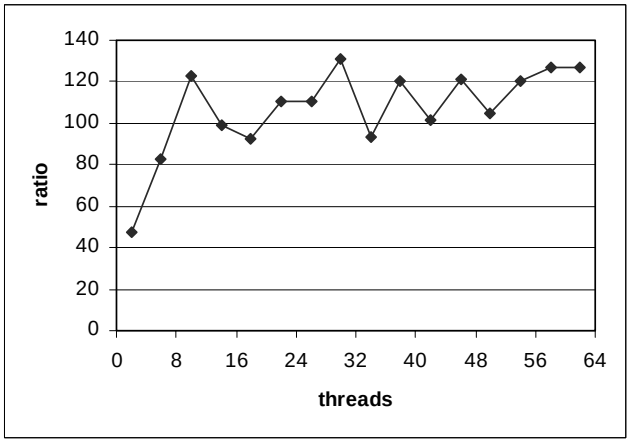

Figure 3: Comparison of futures and threads

code simply returned immediately. We measured the time to complete a batch for increasing batch sizes, and compared the time needed to complete a given batch size for the McRT threads against the time needed to complete the same batch size using the futures. As seen in Figure 3, the creation, scheduling, and de-allocation overhead of a McRT thread is between a factor of 40 and a factor of 120 times greater than the equivalent overhead for using futures.

3.2 Large scale CMP evaluation

This section evaluates McRT performance on a large scale CMP platform. Our experiments are based on a cycle-accurate simulator that was used to model a typical large scale CMP system. The simulated platform consists of an array of up to 32 in-order cores, each of which has 4 threads (similar to [48]). Each core will select a different thread each cycle, round-robin, unless the thread is stalled due to a cache miss or if the thread is in the sleep state. The memory hierarchy consists of a 32KB L1 data cache that is shared by the 4 threads on the core, a 2 MB L2 cache that is shared by all the cores, and an off chip L3 cache of size 4 MB. All caches were simulated with an 8-way set associative configuration. The L1 cache access time is 3 cycles, the L2 cache access time is 12 cycles, and the L3 cache access time is 40 cycles. Our simulator performs a cycle accurate simulation of the execution pipeline for all the HW threads, the different caches (including thread interactions), the coherence protocol for the entire memory subsystem, the bandwidth for data transfer between each part of the memory subsystem, and the interconnect to the external memory.

This configuration highlights key differences between large scale CMPs and comparable SMP systems. First, note that the execution resources in a particular core are shared between 4 threads in the large scale CMP system. SMPs are either not hyper-threaded, or else do not share core execution resources to such a high degree. This makes instruction bandwidth more precious in a large scale CMP system. Second, note that a large scale CMP and a SMP have 2 major differences related to the cache hierarchy, which motivates the need for fine-grained interaction between threads:

- **The ratio of the number of HW processors to the total cache size**. A SMP system with 64 HW processors would typically have a total cache size of several tens or even hundreds of megabytes. SMPs generally comprise of cores optimized for single thread performance, and therefore have a large cache per thread. In contrast, this CMP with 64 HW threads has a total cache size of less than 10 MB. In a CMP, the aggregate cache is significantly reduced because the aggregate die area is significantly reduced (one chip vs. the many chips in a SMP).
- **The relative latency of accessing data from a different thread.** In a SMP system, accessing data from another thread typically implies die to die communication. In a CMP system, latency to another thread's data is orders of magnitude lower since the transfer can be handled without leaving the die. Moreover, with each core in the CMP having multiple threads, the *effective* latency is further reduced for 2 reasons: a) if 4 threads are interleaved in the pipeline, then each operation effectively has $1/4^{th}$ the latency, and b) if a thread in a core is stalled, the other threads in the core can fully consume the core resources.

3.2.1 Effect of instruction bandwidth

This section discusses how we addressed the effect of the highly threaded core architecture on application scalability. Figure 4 shows XviD performance on a single core. The "IPC scaling" bar shows how the instructions per cycle (aggregate instructions for all the threads on a core) increases as we add threads; for example the IPC at 4 threads is 2.5X that of a single thread. The "base" bar shows the performance improvement for XviD running on the baseline McRT; for example, performance at 4 threads is 2X the performance of a single thread. In the baseline

McRT configuration, when a thread is blocked (at a lock, barrier, etc.), the underlying HW thread still executes instructions, thus unnecessarily consuming HW resources. We then added an efficient user-level *Mwait* instruction [51] to the simulator. This instruction allows a software thread to communicate to the processor that it is blocked; in turn, the processor suspends the execution of the corresponding HW thread. The SW thread also passes the address of a location to be monitored for writes; the processor starts execution of the underlying HW thread when it sees a write to the monitored location, or after a specified interval. Thus, Mwait ensures that a blocked thread does not take away HW resources from an executing thread. The "Mwait" bar shows the speedup as we add threads to a core. Note that the application speedup almost exactly matches the IPC increase showing that the software stack is almost perfectly efficient. It is important to note that the addition of Mwait does not reduce the number of blocked cycles; it only ensures that a blocked thread does not waste execution resources. Hence all the performance improvement in Figure 4 arises only from a better utilization of the core instruction bandwidth. ***A SMP system would have a much lower sensitivity to instruction bandwidth.***

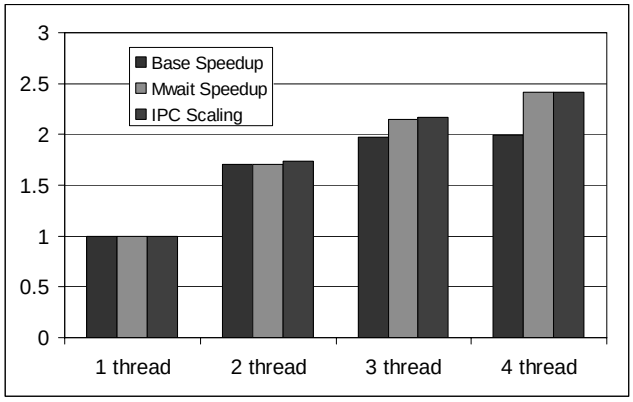

Figure 4: Single thread XviD simulation

When running on the large scale CMP simulator, McRT uses *Mwait* in all the synchronization constructs. For example, if a thread fails to acquire a lock, it uses *Mwait* and passes in the lock address to be monitored. Therefore, the lock release automatically wakes up the thread since the release requires a write into the lock location.

3.2.2 Application scalability

We now describe our experiences in making the XviD video encoder scale on our many core platform. One way of making the encoding process scalable is to encode several frames in parallel. However, since the cache size on a CMP is not sufficient to hold several frames simultaneously, this strategy does not work.

In order to exploit parallelism, the encoding algorithm is designed such that it partitions the frame into 'k' sub-blocks where 'k' corresponds to the number of threads that may be used for encoding; thus the numbers shown here reflect the speedup in encoding a single frame. ***This reinforces the fact that fine-grained parallelism is important in the large scale CMP design.*** We started with the serial version of the encoder and parallelized it with OpenMP using *mainly parallel* for loops (including nested parallel loops) and reductions. The serial execution time and the single threaded execution time of the parallel version of XviD were practically the same.

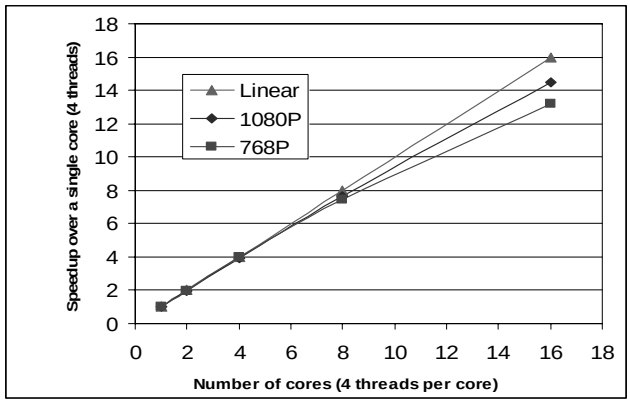

Figure 5: XviD scaling on McRT

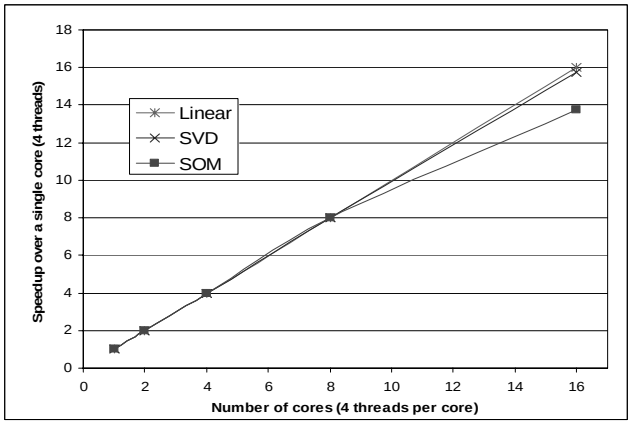

Figure 6: RMS Scalability on McRT

Figure 5 shows the scalability of the XviD encoder on McRT. The x-axis shows the number of cores. Note that for *k* cores the number of HW processors available is *k**4. In this chart, speedup is computed with respect to the execution time on a single core (i.e. the execution time with 4 threads). The figure shows the speedup for 768P (1024x768) and 1080P (1920x1080) frames. This experiment demonstrates that we get almost linear speedup up to configurations that have 64 HW processors. The 1080P shows a slightly better scalability than the 768p since the frame size is larger; hence each thread works on a bigger sub-block and is able to better amortize the threading overhead. The "Linear" line in the graph represents perfect speedup.

Figure 6 shows the speedup seen for the RMS workloads on our many core platform. Once again, these experiments demonstrate that McRT can scale almost linearly up to 64 HW threads. We plot the number of cores on the x-axis with each core containing 4 HW threads. We compute the speedup with respect to the execution time on a single core (4 threads). Again, the serial version of the workloads had essentially the same execution time as the single-threaded parallel version.

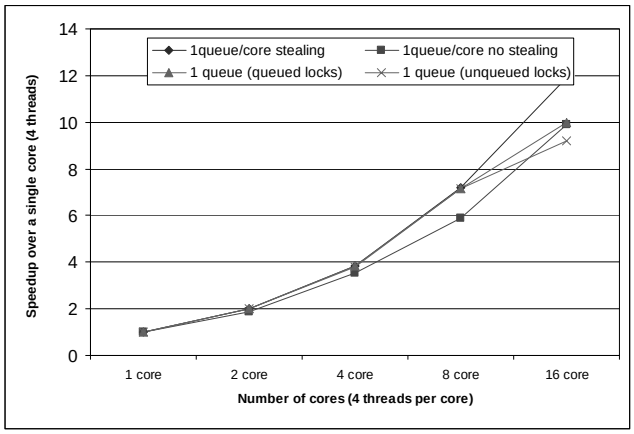

Figure 7: Effect of different scheduling and locking policies

3.2.3 Configurability

This section demonstrates the need for configurability in a large scale CMP runtime. For the sake of discussion, this section primarily concentrates on experiences with XviD.

Figure 7 shows the performance of XviD with different McRT policies. The figure compares the performance with a single scheduling queue using a TTS (test & test & set) lock, a single scheduling queue using a ticket lock, using 1 scheduling queue/core (ticket lock) without any work stealing, and using 1 scheduling queue/core (ticket lock) with work stealing. There is no perceptible difference between the policies till 16 HW threads (4 cores). At 32 HW threads, the distributed scheduling queue (1 queue/core) without work stealing performs the worst. The distributed queue reduces contention; however the absence of work stealing hurts load balance. By definition, a single scheduling queue provides perfect load balance. At 32 HW threads, the load imbalance hurts more than the contention.

On the other hand, at 64 HW threads, contention becomes more pronounced, and mitigates the effect of load imbalance: thus the distributed scheduling queue performs as well as the single queue. The effect of contention is highlighted further by the fact that the queued locking (ticket lock) starts to help compared to a TTS lock. Not surprisingly, the distributed queue with work stealing provides the best result – the load imbalance is removed due to the stealing, and the contention is reduced since the queues are distributed.

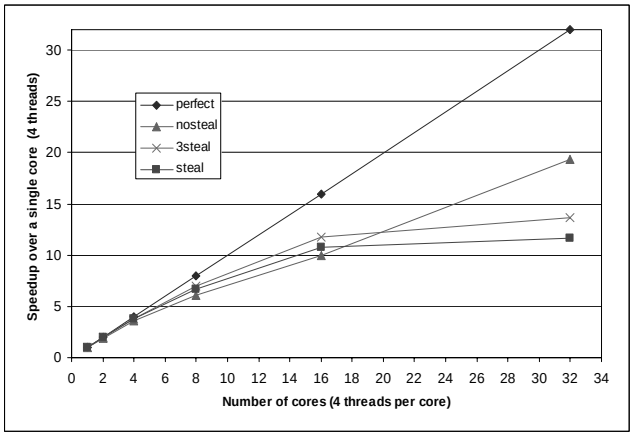

Figure 8: Effect of scheduling on XviD speedup

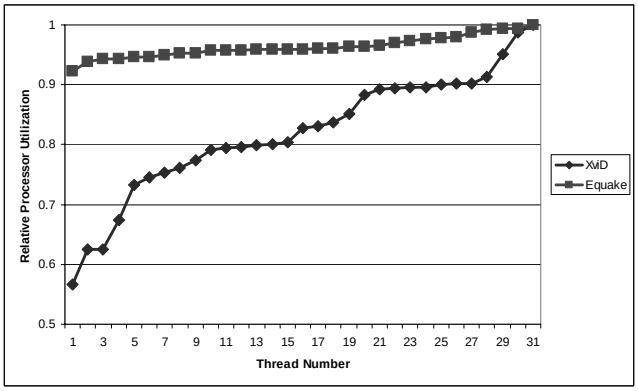

Figure 9: Per thread load balancing in Xvid and Equake

Based on this result, it would appear as though work-stealing is the optimal policy for XviD. Figure 8 shows the XviD speedup as we increase the number of HW threads to 128. At 128 HW threads, the performance of work stealing is worse than that of a no stealing configuration; upon inspection it turned out that work stealing introduces an overhead that grows more than linearly with the number of HW threads. At 128 threads, the overhead of stealing hurts more than the load imbalance. To address this, we added a restricted form of stealing where a processor is allowed to steal work only from the queues of its neighbors (denoted as "3steal" in the graph). The restricted stealing performs better than the unrestricted stealing at all data points, but even that does not perform as well as with no stealing in a 128 HW thread configuration. ***This shows that a runtime system must be configurable to enable different policies to optimize application performance.***

Given an image frame, the amount of computation required for encoding different parts of it can vary significantly. For example, the parts of the frame comprising the background take significantly less computation since they remain static between frames. On the other hand, the parts of the frame containing objects in motion require more computation. Other application domains do not exhibit such imbalance. Figure 9 compares the load balance between XviD and

SpecOMP Equake by comparing the CPU utilization per hardware thread. We use Equake as an example of a HPC-style computationally intensive workload. We used the Spec input set for Equake, and the 1080P frame for XviD.

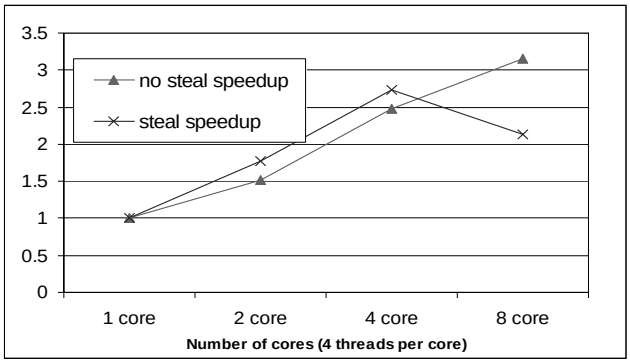

Figure 10: Equake speedup and scheduling policy

We computed the number of instructions executed by different threads when both XviD and Equake were executed with 32 threads. Figure 9 shows the result with each XviD thread's instruction count normalized to that of the (XviD) thread with the maximum instructions; the Equake data was similarly normalized. For XviD, the busiest thread executes almost 2X the instructions of the thread with the least instructions, but for Equake it is only about 10% more. The XviD data reinforces Figure 7 which showed that work stealing helps XviD at 32 threads. The Equake data is reinforced by Figure 10 which shows the effect of work stealing; at 32 threads, stealing performs worse than no stealing. The crossover happens earlier for Equake than for XviD since the load imbalance is less severe.

3.2.4 Synergy between components

As mentioned earlier, a key insight from our experience with implementing McRT was that the individual components could leverage the integrated design to build up a useful synergy. We conclude this section by using the memory manager to quantitatively illustrate this.

We implemented non-blocking and lock-based (blocking) versions of our memory manager. We compared the two versions of the memory manager using the well-known Larson benchmark[50]]; first by running both versions of the memory manager on top of the McRT scheduler, and then by running the two versions on the Windows (Server 2003) scheduler. Figure 11 compares the performance of the benchmark as we increased the number of SW threads on a 8 processor (16 HW threads) IBMx445 machine. As the number of SW threads increases, the lock-based version on the Windows scheduler expectedly degrades due to preemption inside critical sections. As also expected from cooperative scheduling without yields inside critical sections, the lock-based version does not degrade on McRT as the number of SW threads increases; moreover, it also outperforms the non-blocking version on the Windows scheduler once the number of software threads exceeds the number of hardware threads.

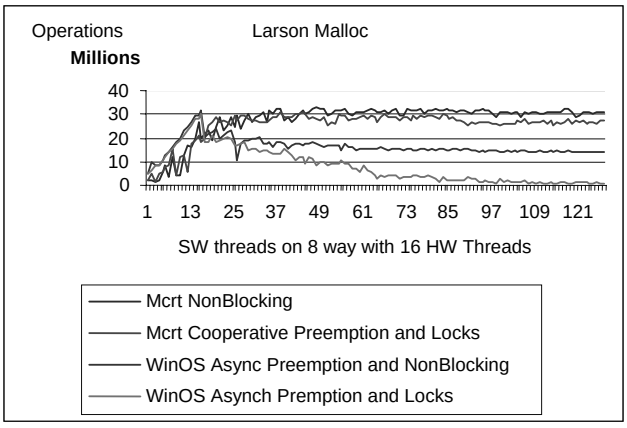

Figure 11: Co-operative scheduling and malloc

This experiment provides a good illustration of the benefits of a holistic design. The non-blocking version requires very sophisticated algorithms as compared to the blocking version. A stand-alone memory manager may opt for the non-blocking implementation; but when integrated with a scheduler the blocking version provides the same properties with a much simpler algorithm.

4. McRT Programmability Enhancements

The main programmability enhancement in McRT is a software transactional memory (STM) library to enable transactional programming [35]. To enforce atomicity, the STM library uses strict two phase locking for writes and optimistic read concurrency. It uses an in-place update scheme and maintains an undo log to rollback aborted transactions. The STM provides both object-level and cache line level conflict detection, and supports nested transactions with partial rollbacks. Since the STM module is not the primary focus of this paper, we refer the reader to [37] for a complete description. We will briefly show how it enhances programmability, and focus mainly on how it leverages the synergy of the overall system.

Figure 12 illustrates the programmability and scalability that is enabled through the use of STM instead of lock-based programming.. This figure compares the performance of concurrent AVL tree operations (80% lookup, and 20% updates) using locks and transactions. Since the tree gets rebalanced on an update, the rebalancing locks the root of the tree and hence the locking version does not scale. The STM algorithm simply replaces the lock acquire by a transaction begin, and the lock release by a transaction end, thus using

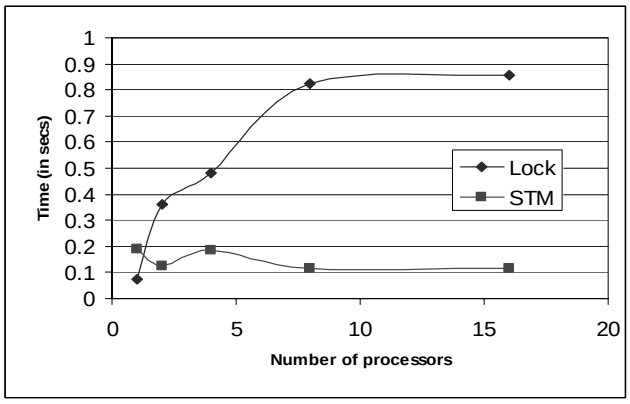

Figure 12: Comparison of STM and locking on AVL tree

exactly the same degree of coarseness in the synchronization. However, the STM performs much better than the locking algorithm since it benefits from read-sharing and the automatic extraction of fine-grained concurrency by the STM library. Note that this performance benefit comes along with the other benefits of transactional programming such as deadlock freedom, etc. The reader may refer to [37] for more comparisons of lock-based and transactional programming.

The STM module is a great example of how synergy plays a key role in the design of McRT. The remainder of this section describes how the STM module leverages the remaining parts of McRT.

STM and memory allocator: The STM module relies on a tight integration with the memory manager to communicate both transaction boundaries as well as the set of pointers accessible from any executing transaction. While [38] provides a complete description of the issues involved and the proposed solutions, we illustrate the synergy with a simple example.

The memory allocator must correctly handle memory allocation inside transactional code blocks; for example, all memory allocated inside an aborted transaction should get deallocated. Moreover, if a nested transaction gets partially rolled back, only the allocations inside the nested transaction must be undone. Second, our STM implements optimistic read concurrency. As shown in [37] this improves scalability by an order of magnitude. However, this also imposes constraints on the allocator. Consider the code sequence in Figure 13 where the function traverses a list to find a node with a given key and replace it.

If we replace the transaction with critical sections, the code would work fine. However with optimistic transactions, multiple threads may simultaneously try to delete a node

```
nodeDelete(int key) {
    ptr = head of list;
    transaction {
        while( ptr->next->key != key ) {
            ptr = ptr->next;
         } /* end while */
     temp = ptr->next;
     ptr->next = ptr->next->next;
    } /* validate & end transaction */
    free(temp); /* Anyone using temp? */
}
```

Figure 13: Optimistic concurrency and deallocation

with a given key. Therefore, multiple threads could get a pointer to the same node (same value of temp). One of the transactions is going to commit, while the remaining will ultimately abort. The committing transaction unlinks the node, but unfortunately does not know when it is safe to free the node since some active transaction may still have a pointer to it. Therefore, the memory allocator needs to work closely with the STM module to determine when it is safe to recycle memory.

STM and scheduler: Almost all current implementations of software transactional memory use non-blocking primitives, which imposes a significant performance overhead. Non-blocking primitives are used to provide progress guarantees in the face of preemption. ***In contrast, the McRT-STM module uses a lock-based implementation to avoid the overhead of non-blocking mechanisms, and leverages the tight integration with the scheduler to provide progress properties.*** Since the co-operative scheduling in McRT makes all preemption points explicit, the STM and the scheduler work together to provide 2 properties: (1) if a thread yields the HW processor in the middle of a transaction, then we set state in the thread's control block to indicate this. Other threads can inspect this state, and abort a transaction only if it has yielded the processor. (2) if a thread gets scheduled back in the middle of a transaction, it first resets the state in the thread's control block to indicate it is active, and then checks whether it has been aborted before resuming the transaction. As we show in [37], this provides a significant performance improvement while still maintaining preemption safety.

Implementing a high performance transactional memory module requires tight integration with compiler optimizations, and a source language *transaction* construct. The McRT-STM library exports an interface to enable this – we have used this interface to integrate with a Java compiler that translates a language-level *transaction* construct and performs aggressive optimizations on transactional code blocks [36], as well as with a garbage collector.

5. Reconfiguring the System Stack Through McRT

In this section, we will discuss the McRT support for heterogeneity, our final requirement for a large scale CMP software stack. In particular, we will focus on our experiences with using McRT to explore a futuristic configuration of the software stack on large scale CMP platforms..We believe enabling such a software stack (also referred to as *sequestered mode* execution) is one of the key motivations for supporting heterogeneity. We will use a case study of compute intensive workloads as are seen in the gaming domain; we used Equake from the Spec2000 suite of benchmarks as a representative compute intensive workload. The sequestered system was set up on an IBM x445 8 way SMP system.

In the sequestered mode, McRT is used as a light-weight threading system that runs on "*bare metal*" replacing the operating system on a subset of the cores. In this configuration McRT provides some of the services, such as scheduling and threading, traditionally provided by an OS. The sequestered cores act as an accelerator where the compute intensive parts of an application can be offloaded. The OS is still responsible for controlling access to the sequestered cores and enforcing security policies. Figure 14 illustrates how the system is organized. The underlying hardware platform is partitioned into 2 domains. The operating system runs on one domain (called the host domain), and McRT runs on the other domain (called the sequestered domain) without an underlying OS. Note that, in theory, there is no restriction on the number of domains that can be created. As discussed before, McRT internally abstracts the underlying execution resources as *logical processors*, and lets an application control affinity, work-dealing, and other policies with respect to the logical processors. In the host domain, McRT maps the logical processors to kernel threads; but in the sequestered domain, McRT maps the logical processors directly to the sequestered processors. This allows for an application to directly control and co-ordinate the underlying hardware.

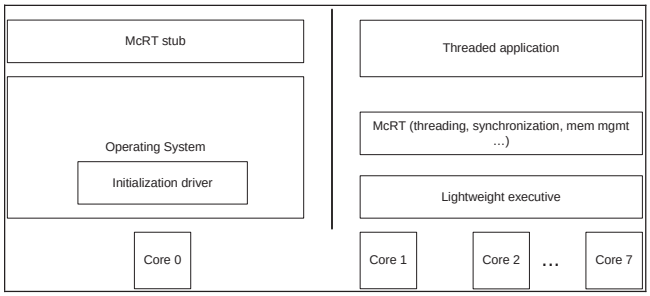

Figure 14: Sequestered mode execution

In our implementation, during bootup, MS Windows Server 2003 is loaded on one processor, while the other seven start up in sequestered mode. McRT starts up on the host processor, and then starts executing on the sequestered processors. The McRT task queue mechanism is used to realize sequestered partitioning. Specifically, two task queues are set up– one of which (Q_H) is snooped by the host processor, the other (Q_S) is snooped by the remaining sequestered processors. McRT adds the starting application thread – the one that will execute the application "main" – to the sequestered queue Q_S. In this configuration McRT is set up not to perform any work-sharing or work stealing. Hence, any threading abstractions created by the application are added to the sequestered queue, and executed only by the sequestered processors. Similarly, any abstractions in the host queue are executed only by the host processors, thus maintaining two separate execution domains.

The sequestered domain contains a lightweight executive that provides only basic interrupt handling features; all of the threading services such as scheduling, synchronization, etc. are provided by McRT. During startup, the OS allocates and pins the memory for the sequestered cores, which prevents any page faults in the sequestered cores. The lightweight executive includes a memory manager that allocates memory out of the pinned pages. All basic interactive services such as I/O are provided by the host domain. The sequestered domain intercepts all service requests and passes them onto the host domain. In the current implementation, the application is recompiled so that the system/libc calls get re-vectored to wrappers implemented by McRT. (e.g. a call to `fopen` would get translated into `sequestered_fopen` implemented in McRT). The system call wrappers leverage the scheduler API to cause the host processor to execute the call. For example consider the following wrapper:

```
sequestered_fopen(char* path, char* mode) {
  FILE* fptr;
  switchToQueue(hostQueue);
  fptr = fopen(path, mode);
  switchToQueue(sequesteredQueue);
  return fptr;
}
```

The first `switchToQueue` operation causes the current task to yield the logical sequestered processor and add itself to a host queue. A host processor (in our case, the single host processor) will pick up the task, execute the system call, and return the task to a sequestered queue through the second `switchToQueue` operation.

Figure 15 shows how Equake performs on the sequestered system. We first ran Equake on the 8 way system by having Windows run on all the processors – a conventional SMP setup. These numbers are reported as "Native" and "McRT-OS" – "Native" refers to the performance from using the Intel OpenMP implementation on Windows, and McRT-OS refers to the performance from using McRT running on top of Windows on the 8 way SMP. McRT-Sequestered refers to the performance from using McRT in

sequestered mode with a single host processor and 7 sequestered processors. All speedups are computed with respect to the "Native" single thread execution time. Note that the sequestered mode execution is much more efficient than running on top of an OS.

The sequestered mode emulates several features of a futuristic software stack. First, the application execution characteristics are different in the two domains: the application performs computation on the sequestered processors, and system calls on the host processors. Clearly, this can be generalized to other scenarios: e.g. serial code on the host processor, and parallel code on the sequestered processors. Second, an application can control the utilization of hardware resources in a fine-grained manner by using the switchToQueue operation. In our setup, the

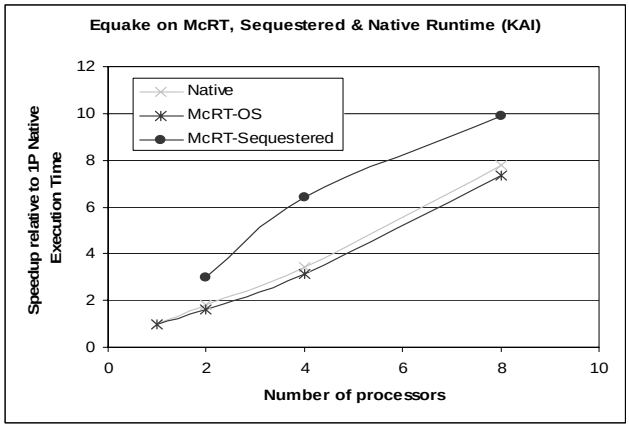

Figure 15: Equake on sequestered system

application uses it to control only the execution of system calls, but clearly this can be generalized to enable the execution of different code regions in different domains.

The sequestered mode also benefits from the tight integration among the McRT components. The McRT OpenMP adapter leverages the OpenMP fork-join parallelism model to automatically detect serial code regions. We added calls in our OpenMP adapter to switch to the host domain in the serial portions, and switch back to the sequestered domain in the parallel portions. Such optimizations would be very helpful if the underlying HW were heterogenous – for example, an application could automatically execute the serial portions on cores with good scalar performance, and parallel portions on throughput oriented cores.

We believe that these experiments are good preliminary design points for evaluating how operating systems should be structured in future many-core platforms. There are several issues such as isolation, partitioning, etc, that are topics for future research. For example, our current partitioning scheme is biased towards serving compute intensive workloads, but the heuristics may need to be altered for catering to I/O intensive workloads, potentially allowing for configurations such as [11] on chip..

6. Related Work

Our work was inspired by previous projects in the areas of language systems, operating systems design and high performance computing. Currently, operating systems such as Linux and MS Windows treat CMP architectures as a SMP on chip[5][6]. This paper shows why such an approach does not scale in the large-scale CMP context.

The threading and synchronization primitives provided by McRT are similar to those seen in traditional user level threading packages such as pthreads[2], NGPT[3], NPTL[4] and Capriccio[8]. These primitives can be used to realize a variety of parallel execution models including those seen in traditional thread-based and event-based programs [10][11][12][13]. The McRT scheduler is comparable to user level schedulers, that have been explored exhaustively in the context of micro-kernels and customizable operating systems such as L4[14], Exokernel[15], Flux[16], and SPIN[29]. In comparison, the McRT scheduler is more light-weight, provides a high degree of configurability, and can be tightly integrated with language systems [47].

Several operating systems have explored issues related to performance and scalability on multi-processor platforms [1][17][18][19][20]. While the McRT framework is similar in spirit to these projects and learnt from their experiences, it is more lightweight and had to address significant differences in the underlying hardware. We believe the McRT framework can provide an evolutionary pathway for the redesign of operating systems to cater to large scale CMP architectures. This work is also related to projects such as Disco[21] and K42[22][23] that explore scalable operating systems. The primary difference is that our system is restricted to shared memory multi-processing that will be typical on futuristic CMP platforms.

Given the scale of parallelism, our work is also related to previous research in high performance computing. Light-weight kernels such as PUMA & Cougar [24][25][26] can be considered as closely related projects. In general, as the number of available processing units increases to 10's and 100's, we believe that the system structures will move closer to these light-weight configurations. Even though McRT in sequestered mode is similar to the above systems, there are still two primary differences: first, light weight kernels are typically customized for single application domains, but McRT provides a generic configurable framework that can easily be customized at different levels to achieve performance and scalability; second, there are significant architectural differences between the hardware platforms and it is not obvious if light weight kernels can directly scale to handle this level of parallelism on a single chip (single node). Finally, the Piglet system [53] is also

comparable to McRT in sequestered mode.. The sequestered mode software stack can be thought of as complementary to ongoing research for using hyper-visors and virtual machine monitors[27][28] to address scalability on multi-processor platforms.

The McRT framework is also comparable to previous work in building language runtimes such as for CILK[30], and OpenMP[31]. While McRT provides several features that enable tight integration with programming language models, it is platform-neutral and thus provides the added benefit of portability in addition to customizability and configurability. Finally, the specific mechanisms used in the McRT-STM and the McRT memory manager have been described elsewhere [37][38], while the compiler integration has been discussed in [36].

7. Conclusions

This paper presented the design and implementation of McRT, the first runtime system targeted at futuristic large scale CMP platforms. We have shown how the large scale CMP platform introduces several novel challenges that directly affect the design of a system software stack. McRT's design was based on a holistic re-evaluation of the different components that make up a software stack. These components were tightly integrated with each other to enable good performance and scalability. Experimental evaluation demonstrates how McRT can be used to scale almost linearly to 64 HW threads on workloads that represent emerging usage models for large scale CMP platforms. Finally, we have shown how McRT is being used to reevaluate the role of the OS on large-scale CMP systems, and presented initial encouraging results for such a system design.

8. Acknowledgements

We would also like to thank Thomas Gross and the anonymous reviewers for their comments which helped to strengthen the paper.

9. REFERENCES

[1] B. D. Marsh, M. L. Scott, T. J. LeBlanc, and E. P. Markatos. Firstclass user-level threads. Proc. SOSP-13, October 1991.

[2] B. Lewis and D. J. Berg, "Multithreaded Programming with Pthreads," Prentice Hall, 1998.

[3] Next Generation POSIX Threading. http://www-124.ibm.com/pthreads/

[4] U. Drepper, and I. Molnar. The native POSIX thread library for Linux, Jan 2003. http://people.redhat.com/drepper/nptl-design.pdf.

[5] D. Vianney, Hyper-Threading speeds Linux, Jan 2003. http://www-128.ibm.com/developerworks/linux/library/l-htl/

[6] Microsoft Corp, Windows Support for Hyper-Threading technology, 2002. download.microsoft.com/download/5/7/7/577a5684-8a83-43ae-9272-ff260a9c20e2/Hyper-thread_Windows.doc

[7] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. Proc. SOSP-16, 1997.

[8] R. von Behren, J. Condit, F. Zhou, G. C. Necula, and E. Brewer, "Capriccio: Scalable threads for internet services," Proc. SOSP-19, 2003.

[9] N. Nagarajaya, Improving Application Efficiency Through Chip Multi-Threading, Sun Developer Network, Mar 2005. developers.sun.com/solaris/articles/chip_multi_thread.html

[10] V. S. Pai, P. Druschel, and W. Zwaenepoel. Flash: An efficient and portable Web server. In Proceedings of the USENIX Technical Conference, Monterey, CA, June 1999.

[11] M. Welsh, D. Culler, and E. Brewer. "SEDA: An Architecture for Well-Conditioned, Scalable Internet Services." Proc. SOSP-18, 2001.

[12] S. Gribble, M. Welsh, R. von Behren, E. Brewer, D. Culler, N. Borisov, S. Czerwinski, R. Gummadi, J. Hill, A. Josheph, R. Katz, Z. Mao, S. Ross, and B. Zhao. The Ninja Architecture for Robust Internet-Scale Systems and Services. Sp. Iss.: Computer Networks on Pervasive Computing 2000.

[13] Mohan Rajagopalan, Saumya Debray, Matti Hiltunen, and Richard Schlichting. Profile-directed optimization of event-based programs. Proc. PLDI, 2002

[14] J. Liedtke, On micro-Kernel Construction, SOSP-15, 1995.

[15] D. R. Engler, M. F. Kaashoek, and J. O'Toole Jr. Exokernel: an operating system architecture for application-specific resource management. SOSP-15, 1995.

[16] B. Ford, G. Back, G. Benson, J. Lepreau, A. Lin, and O. Shivers. The Flux OSKit: A Substrate for Kernel and Language Research. SOSP-16, 1997.

[17] J. Ousterhout, A. Cherenson, F. Douglis, M. Nelson, and B. Welch. The Sprite network operating system. IEEE Computer, 21(2):23--36, February 1988.

[18] Anderson, T. E., Bershad, B. N., Lazowska, E. D., and Levy, H. M. Scheduler Activations: Effective Kernel Support for the User-Level Management of Parallelism. ACM ToCS, Feb. 1992

[19] T. Anderson, E. Lazowska, and H. Levy. The Performance Implications of Thread Management Alternatives for Shared-Memory Multiprocessors. IEEE Trans. on Comp.,Dec. 1989.

[20] Michael B. Jones, Richard F. Rashid: Mach and Matchmaker: Kernel and Language Support for Object-Oriented Distributed Systems. OOPSLA 1986

[21] E. Bugnion, S. Devine, and M. Rosenblum. Disco: running commodity operating systems on scalable multiprocessors. SOSP-16, 1997.

[22] The K42 project, IBM Research. http://www.research.ibm.com/k42/

[23] The K42/Tornado Operating System. http://www.eecg.toronto.edu/~tornado/

[24] T. G. Mattson and G. Henry. An overview of the Intel TFLOPS supercomputer. Intel Technology Journal, 1, 1998.

[25] Sharad Garg, Robert Godley, Richard Griffiths, Andrew Pfiffer, Terry Prickett, David Robboy, Stan Smith, T. Mack Stallcup, and Stephen Zeisset. Achieving large scale parallelism through operating system resource management on the Intel TFLOPS supercomputer. Intel Technology Journal, 1st quarter 1998.

[26] Ron Brightwell, Rolf Riesen, Keith D. Underwood, Trammell Hudson, Patrick G. Bridges, Arthur B. Maccabe: A Performance Comparison of Linux and a Lightweight Kernel. CLUSTER 2003: 251-258

[27] IBM Research Hypervisor. http://www.research.ibm.com/hypervisor/.

[28] Boris Dragovic, Keir Fraser, Steve Hand, Tim Harris, Alex Ho, Ian Pratt, Andrew Warfield, Paul Barham, and Rolf Neugebauer. Xen and the Art of Virtualization. SOSP, 2003.

[29] Brian N. Bershad, Stefan Savage, Przemyslaw Pardyak, Emin Gün Sirer, Marc E. Fiuczynski, David Becker, Craig Chambers, Susan J. Eggers: Extensibility, Safety and Performance in the SPIN Operating System. SOSP 1995.

[30] R. D. Blumofe, C. F. Joerg, B. C. Kuszmaul, C. E. Leiserson, K. H. Randall, and Y. Zhou. Cilk: An Efficient Multithreaded Runtime System. PPoPP, 1995.

[31] H. Lu, Y. C. Hu, and W. Zwaenepoel. OpenMP on networks of workstations. In Supercomputing '98, November 1998

[32] J. Rattner. Platform 2015. Intel Dev. Forum, Spring 2005.

[33] J. Rattner. Tera-Scale Research Program. Intel Dev. Forum, Spring 2006.

[34] J. Dean, S. Ghemawat. MapReduce: Simplified data processing on large clusters. OSDI, 2004.

[35] M. Herlihy, and J. E. B. Moss. Transactional memory: architectural support for lock-free data structures. ISCA 1993

[36] A. Adl-Tabatabai, B.T. Lewis, V.S. Menon, B.M. Murphy, B. Saha, T. Shpeisman. Compiler and runtime support for efficient software transactional memory. PLDI 2006.

[37] B. Saha, A. Adl-Tabatabai, R. Hudson, C. Minh, B. Hertzberg. McRT-STM: A High Performance Software Transactional Memory System For A Multi-Core Runtime. PPoPP, 2006.

[38] R. Hudson, B. Saha, A. Adl-Tabatabai, B. Hertzberg. McRT-Malloc: A Scalable Transaction Aware Memory Allocator. ISMM, 2006.

[39] E. Grochowski, R. Ronnen, J. Shen, H. Wang. Best of both latency and throughput. ICCD, 2004.

[40] Cierniak, M., Eng, M., Glew, N., Lewis, B., and Stichnoth, J. 2005. The Open Runtime Platform: a flexible high-performance managed runtime environment: Research Articles. Concurr. Comput. : Pract. Exper, Apr. 2005.

[41] P. Dubey. Recognition, Mining, and Synthesis moves computers to the era of tera. *Technology@Intel*, Feb 2005.

[42] Craig, T. S. Building FIFO and priority-queueing spin locks from atomic swap. Technical Report TR 93-02-02, Dept of Computer Science, University of Washington, Feb. 1993.

[43] Magnussen, P., A. Landin, and E. Hagersten. Queue locks on cache coherent multiprocessors. *8th Intl. Parallel Processing Symposium*, Cancun, Mexico, Apr. 1994.

[44] M. L. Scott and W N. Scherer III. Scalable Queue-Based Spin Locks with Timeout. PPoPP 2001.

[45] Doug Lea. The java.util.concurrent Synchronizer Framework. *Science of Computer Programming*, Dec 2005.

[46] A. W. Appel. Compiling with continuations. Cambridge University Press, New York, 1992.

[47] B. So, A.M. Ghuloum, Y. Wu. Optimizing data parallel operations on many-core platforms. STMCS 2006.

[48] P. Kongetira, K. Aingaran, K. Olukutun. Niagara: A 32-way Multithreaded Sparc Processor. IEEE Micro, Mar 2005.

[49] J. M. Mellor-Crummey and M. L. Scott. Algorithms for scalable synchronization on shared-memory multiprocessors. ACM Transactions on Computer Systems, 9(1):21--65, 1991

[50] P. Larson and M. Krishnan. Memory Allocation for Long-Running Server Applications. In *Proceedings of the First International Symposium on Memory Management*, pages 176–185, Vancouver, BC, October 1998

[51] IA-32 Intel Architecture Software Developer's Manual. Intel Corporation.

[52] Bradski, G.; Kaehler, A.; Pisarevsky, V. "Learning-Based Computer Vision with Intel's Open Source Computer Vision Library." Intel Technology Journal. http://developer.intel.com/technology/itj/2005/volume09issue02/art03_learning_vision/p01_abstract.htm. May 2005.

[53] Muir, S. and Smith, J. 1998. Functional divisions in the Piglet multiprocessor operating system. In *Proceedings of the 8th ACM SIGOPS European Workshop on Support For Composing Distributed Applications* (Sintra, Portugal).