A Coherent and Managed Runtime for ML on the SCC

KC Sivaramakrishnan†, Lukasz Ziarek♮, and Suresh Jagannathan†
† Purdue University, ♮ SUNY Buffalo
{chandras, suresh}@cs.purdue.edu, lziarek@buffalo.edu

***Abstract*—Intel's Single-Chip Cloud Computer (SCC) is a many-core architecture which stands out due to its complete lack of cache-coherence and the presence of fast, on-die interconnect for inter-core messaging. Cache-coherence, if required, must be implemented in software. Moreover, the amount of shared memory available on the SCC is very limited, requiring stringent management of resources even in the presence of software cache-coherence.**

In this paper, we present a series of techniques to provide the ML programmer a cache-coherent view of memory, while effectively utilizing both private and shared memory. To that end, we introduces a new, type-guided garbage collection scheme that effectively exploits SCC's memory hierarchy, attempts to reduce the use of shared memory in favor of message passing buffers, and provides a efficient, coherent global address space. Experimental results over a variety of benchmarks show that more than 99% of the memory requests can be potentially cached. These techniques are realized in MultiMLton, a scalable extension of MLton Standard ML compiler and runtime system on the SCC.

I. Introduction

One of the key benefits, and perhaps, the most interesting aspect of the Intel Single-chip Cloud Computer (SCC) [1] is the lack of cache-coherence and the presence of a fast, hardware interconnect for message passing. This design choice, however, forces the programmer to treat the SCC as a cluster of machines and to be programmed in an SPMD style. Programs written for an SMP system under the assumption of coherent shared memory, unfortunately, cannot be easily ported to the SCC. Such programs not only must be re-designed, but must strictly manage the use of non cache-coherent shared memory for inter-core communication due to its size. Efficient use of the SCC's message passing buffers (MPB) is often required for scalability and to ease memory pressure.

As such, any programming model which allows programming processors like the SCC in a manner similar to a traditional cache-coherent multicore machine, by hiding the architectural details of the SCC, must provide: 1) software based cache-coherence, 2) memory management for shared memory, and 3) use of MPBs for inter-core communication. Thus, the runtime should perform the task of effectively mapping the uniform memory space onto the private, core-local memories, as well as the shared memory. In addition, the model should also hide the existence of specialized messaging hardware like the MPBs. Whenever profitable, inter-core communication should be automatically performed through MPBs, without relying on the programmer to identify such communication.

To achieve such a programming model, we introduce a series of compiler, garbage collection, and runtime techniques. First, we introduce a garbage collection scheme with thread-local heaps and shared heaps to provide a single coherent address space across all of the cores. The runtime automatically optimizes locality of objects based on the type information, such that, most object accesses are cached. Second, whenever possible, inter-core communication is optimized using object type information to eschew usage of shared memory for fast, on-die MPB memory. These techniques are realized in the MultiMLton compiler and runtime system for a parallel extension of Standard ML.

This paper presents the following contributions:

- A memory management scheme for the SCC which provides an efficient, global, coherent address space
- A thread-local garbage collector that uses object type information and software managed coherence for efficient object placement
- A type discriminated mapping of first-class channel operations on to the MPB memory

II. Background

A. MultiMLton

MultiMLton is a scalable, whole-programming optimizing compiler and multicore aware runtime system for Standard ML [2], a mostly functional language. Parallel programs in MultiMLton are written using ACML [3], an asynchronous extension of Concurrent ML [4]. Concurrency in ACML is expressed in the form of lightweight user-level threads, and typical ACML programs create thousands of threads during their lifetime. The lightweight thread allows the programmer to reason about concurrency as a design principle rather than as a way to extract parallelism from the program. It is the duty of the runtime system to schedule, manage and load balance the lightweight threads across multiple cores in order to exploit the parallelism on the target platform.

Although MultiMLton supports mutable state, the programmer is encouraged to share data between threads using message-passing, and restrict the use of mutable state within each thread. To this end, MultiMLton has excellent support for constructing composable message passing protocols. Threads in MultiMLton can communicate with each other by sending messages, both synchronously as well as asynchronously, over first-class typed channels. Apart from sending immutable values, mutable ref cells, arrays and channels may also be

exchanged over channels. In addition, complex communication protocols can be successively built up from smaller kernels in a monadic style using *events*.

B. SCC Processor

Intel's Single-chip Cloud Computer (SCC) is an experimental platform from Intel labs with 48 P54C Pentium cores. The most interesting aspect of SCC is the complete lack of cache coherence and a focus on inter-core interactions through a high speed mesh interconnect. The cores are grouped into 24 tiles, connected via a fast on-die mesh network. The tiles are split into 4 quadrants with each quadrant connected to a memory module. Each core has 16KB L1 data and instruction caches and 256KB L2 cache. Each core also has a small message passing buffer (MPB) of 8KB used for message passing between the cores. Programmers can either use MPI or RCCE, a lightweight message passing library tuned for SCC [5].

Since the SCC does not provide cache coherence, coherence must be implemented in software if required. From the programmer's perspective, each core has a private memory that is cached and not visible to other cores. The cores also have access to a comparatively small shared memory, which is by default not cached to avoid coherence issues. However, caching can be enabled on the shared memory with the caveat that the coherence protocol must be implemented in software. The cost of accessing data from the cached local memory is substantially less when compared to accessing uncached shared memory, since each shared memory access must be routed through the mesh interconnect to the memory controller.

Software managed coherence (SMC) [6] for SCC provides a coherent, shared, virtual memory space on the SCC. It exploits SCC's support for tagging a specific virtual address space as having message passing buffer type (MPBT). Data typed as MPBT bypass L2 and go directly to L1. SCC also provides a special, 1-cycle instruction called `CL1INVMB` that marks all data of type MPBT as invalid L1 lines. By tagging a shared virtual memory space as having MPBT, the programmer can control caching in this virtual address space. However, since SCC does not provide cache coherence, the programmer is recommended to use release consistency model to utilize the cached shared virtual memory. A core issues a `smcAcquire()` to fetch changes from other cores and issues `smcRelease()` to publish its updates.

III. System Design

A. Threading System

In our implementation, we spawn one Linux process per SCC core which serves as a virtual processor (VProc) for the lightweight threads. The lightweight threads are multiplexed over the VProcs. Each VProc runs a scheduling loop that looks for runnable lightweight threads and schedules them. Lightweight threads are preemptively scheduled. Load balancing is achieved by a work sharing mechanism where new lightweight threads are spawned in a round-robin fashion on the VProcs, to which they are pinned for the rest of their execution. Finally, the lightweight threads run concurrently with other threads in the same global address space.

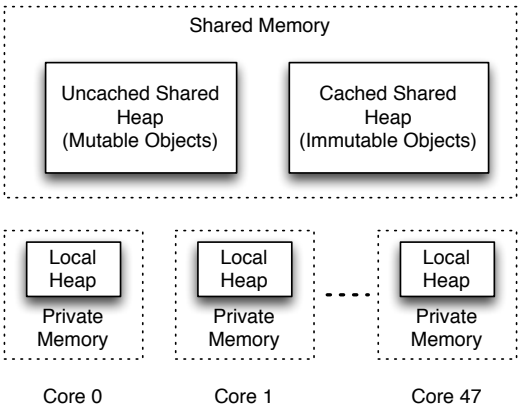


Fig. 1: Heap Layout.

```
1  pointer readBarrier (pointer p) {
2    if (!isPointer(p)) return p;
3    if (getHeader(p) == FORWARDED) {
4      /* Address in shared heap */
5      p = *(pointer*)p;
6      if (p > MAX_CSH_ADDR) {
7        /* Address in cached shared heap, and has not
8         * been seen so far. Fetch the updates. */
9        smcAcquire();
10       MAX_CSH_ADDR = p;
11     }
12   }
13   return p;
14 }
```

Fig. 2: Read barrier.

B. Memory Management

1) Local collector: In order to provide a semblance of a single global address space, we split the program's heap among the set of cores as shown in Figure 1. We call this design a *local collector*. Other terms have also been used in literature including *private nurseries*, *thread-local collector*, etc, [7], [8], [9], [10]. Each core has a local heap into which new objects are allocated. No pointers are allowed from one local heap to another since the other core's private memory is inaccessible.

In order to share data between cores, we also create shared heaps in the shared memory which is accessible to all of the cores. Objects are allocated in the shared heap only when there is an explicit need to share the objects with another core. This occurs when spawning a new lightweight thread on another core, during which time, the environment of the function closure is shared between the spawning and spawned cores. New objects are allocated in the local heap and the shared heaps by bumping the corresponding heap-local frontier.

2) Memory Barriers: Inter-core sharing can also occur when a shared heap object is assigned a reference to a local heap object (exporting writes). During an exporting write, a pointer is created from the shared heap to the local heap. In order to prevent a core from transitively accessing another core's local heap through such a pointer, a *write barrier* is triggered on an exporting write. The pseudo-code for the write

```
1  val writeBarrier (Ref r, Val v) {
2    if (isObjptr(v) &&
3        isInSharedHeap(r) &&
4        isInLocalHeap(v)) {
5        /* Move transitive object closure to shared
6         * heap, and install forwarding pointers */
7        v = globalize (v);
8        /* Publish the updates */
9        smcRelease();
10   }
11   return v;
12 }
```

Fig. 3: Write barrier.

barrier is given in Figure. 3. Before an exporting write, the transitive closure of the local heap object is moved to the shared heaps and then the assignment is performed. We call this *globalization* of local heap objects. Globalization prevents pointers from the shared heaps to the local heaps.

In the place of objects globalized during an exporting write, *forwarding pointers* are installed in the original objects, which now point to their shared heap counterparts. In order to interpret the forwarding pointers, every object read also needs a *read barrier*, which returns the correct address of the shared heap object (Figure. 2). For more information about the local collector and memory barriers, we refer the interested reader to our earlier work [10].

3) Bi-partitioned shared heap: Unlike a typical local heap collector, including our earlier work [10], which has a single shared heap, we split our shared heap into cached and uncached partitions. We take advantage of the fact that standard ML can statically distinguish between mutable and immutable objects. Since immutable objects by definition will not change after initialization, we enable caching on one of the shared heaps into which only globalized immutable objects will be allocated. We call this heap a cached shared heap (CSH). Since most objects in standard ML are immutable, we gain the advantage of caching by placing these objects in CSH while not having to deal with coherence issues. CSH is implemented using Software Managed Coherence (SMC) for SCC [6]. We choose the default (`SMC_CACHE_WTMP`) caching policy for CSH where the data bypasses L2 and caching operates in a write-through mode.

Caching is disabled in the uncached shared heap (USH) into which globalized mutable objects are allocated. By disabling caching, we circumvent the coherence issues at the cost of performance. A local heap object being globalized might contain both mutable and immutable objects in its transitive object closure. Hence, globalization might involve allocating new objects in both partitions of the shared heap. For the same reason, pointers are allowed between the two partitions of the shared heap.

4) Memory Consistency: In order to ensure that the updates to and from CSH are visible to all the cores, explicit cache invalidations and flushes must be implemented in the memory barriers. CSH is always mapped to an address greater than the starting address of the USH. Each core maintains the largest address seen in CSH in the `MAX_CSH_ADDR` variable. During an object read, if the address of the object lies in the shared heap and is greater than `MAX_CSH_ADDR`, we invalidate any cache lines that might be associated with CSH by invoking `smcAcquire()` (Line 9 in Figure 2). This ensures that the values read are not stale. Since the objects in CSH are immutable, there is no need to perform cache invalidation while reading an address that is less than `MAX_CSH_ADDR`. Finally, after garbage collection, `MAX_CSH_ADDR` is set to point to the start of the CSH.

Similarly, whenever an object is globalized to the CSH, we must ensure that the updates are visible to all of the cores. After an exporting write, we invoke `smcRelease()`, to flush any outstanding writes to the memory (Line 9 in Figure 3).

5) Garbage Collection: As mentioned before, the local collector has two invariants. First, no pointers are allowed from one local heap to another and second, no pointers are allowed from the shared heap to any local heaps. Apart from preventing the program from directly trying to access another core's private memory, the heap invariants also allow for independent collection of the local heaps. This is the primary motivation for local collector designs. While the local heaps are collected concurrently, the shared heap is collected after stopping all of the cores. Independent collection of local heaps means that less coordination is required for majority of the garbage collections. This scheme is particularly suited for SCC like architecture where the cores are to be treated as a cluster on a chip.

While the local heap collection proceeds concurrently, the shared heap collection is performed in an SPMD fashion. After the initial barrier synchronization, each processor collects roots from its local heap which is followed by a single core collecting the shared heaps. The garbage collection algorithm is Sansom's dual-mode garbage collection [11], which starts off as a Cheney's copying collector and dynamically switches to Jonker's sliding mark-compact collector.

C. Channel Communication

The primary mode of interaction between lightweight threads is by exchanging messages over first-class typed channels. MultiMLton supports both synchronous and asynchronous channel communication. During a synchronous communication, the sender or receiver thread blocks until a matching communication is available. When a thread blocks, the scheduler switches control to another runnable thread from the queue of threads. When a thread becomes unblocked after a communication match, it is added back to the scheduler queue. Asynchronous communication is implemented by spawning a new parasitic thread [12], which performs the communication synchronously.

Channels are also many-to-many, where multiple sender and receiver threads can communicate over the same channel. The pairing up of a sender with a receiver thread is determined based on their order of arrival. Channels are implemented as a two tuple with a FIFO queue each for blocked senders along with the value being sent and blocked receiver threads, and are treated as regular heap objects.

1) Heap invariant exception – Remembered List: Whenever a thread performs a channel communication, it can get blocked if the matching communication action is not available. A reference to the thread is added to the channel queue such that the thread may be restarted at a later point of time. If the channel happens to be in the shared heap, the thread, its associated stack and the transitive closure needs to be moved to the shared heap to avoid breaking the local collector heap invariants. Since channel communication is the primary mode of thread interaction in our system, we would quickly find that most local heap objects end up being globalized to the shared heap. This would be highly undesirable.

Hence, we allow shared heap channel objects to reference thread objects residing in the local heap. A reference to such a thread is added to a remembered list of objects, which is considered as a root for local heap collection.

2) Communication Optimization: Our channel implementation exploits both the cached shared heap and MPB for efficient inter-core message passing. We take advantage of our heap layout and the availability of static type information to take advantage of the fast, on-die MPB memory. We consider the following five cases:

1) Channel is located in the local heap
2) Channel is located in the shared heap, and the message being sent is an unboxed value
3) Channel is located in the shared heap, the message is in the local heap, and at least one of the objects in the transitive closure of the message being sent is mutable
4) Channel is located in the shared heap, the message is in the local heap, and all objects in the transitive closure of the message being sent are immutable
5) Channel and the message are located in the shared heap

For case 1, we observe that only channels that are located in the shared heap can be used for inter-core communication. Our heap invariants prevent pointers from one local heap to another. Thus, if a channel is located in the local heap, then no thread on the other cores have a reference to this channel. Thus, communication under case 1 only involves a value or a pointer exchange between the communicating lightweight threads.

MultiMLton supports unboxed types that represent raw values. Hence, under case 2, we add a reference to the thread along with the value being sent to the channel. In addition, we add a reference to the blocked thread to the remembered list so that the local garbage collection can trace it.

If the message being sent has a mutable object in the transitive closure, we must make this object visible to both the sender and the receiver core. Figure 4 shows the case where a thread `T1` sends a mutable object `a` on a shared heap channel `C`. In this case, we eagerly globalize `a` before `T1` blocks. Since the message is already in the shared heap, when the receiver thread `T2` eventually arrives, it just picks up a pointer to the message in the shared heap.

Figure 5 shows the case where a thread `T1` sends an immutable object `a` on a shared channel `C`. Here, we simply add to the channel `C`, a reference to the message `a` in the local heap, along with the reference to the thread `T1`. In addition, a reference to the object `a` is added to the remembered list, so that a local garbage collection will be able to identify `a` as being alive.

Afterward, when the receiver thread `T2` arrives and finds the message not to be in the shared heap, it sends an inter-core interrupt to the core on which the message is located (core 0, in this case). After this message transfer is initiated over the MPB using RCCE to transfer the object `a` from core 0 to core 1. Since Standard ML immutable objects do not have identity, making a copy of the immutable object is safe under MultiMLton.

If the channel and the message are located in the shared heap, communication only involves a value or a pointer exchange. This case is similar to case 1.

IV. Evaluation

For our experimental evaluation, we picked 8 benchmarks from the MLton benchmark suite. The benchmarks were derived from sequential standard ML implementation and were parallelized using ACML [3]. The details of the benchmarks and their characteristics can be found at [10]. Our programs partition the available work among thousands of lightweight threads, multiplexed on to the available cores. Hence, although we do not perform load balancing after the threads have started to run, we get a fairly uniform distribution of load across the cores.

The core, mesh controller, and memory on the SCC can be configured to run at different frequencies. For our experiments we chose 533 MHz, 800 MHz, and 800 MHz for core, mesh, and memory respectively.

A. Bi-partitioned shared heap

In order to analyze the performance of our bi-partitioned shared heap scheme, we implemented a local collector with a single shared heap. Since this heap would be used to store both mutable as well as immutable data, caching is disabled on the single shared heap. Figure 6a shows the normalized speedup of the two local collector designs as we increase the number of cores. On 48 cores, the bi-partitioned shared heap collector is 48% faster than the single shared heap collector.

Figure 6b illustrates the space-time trade-offs critical for garbage collector evaluation. The results were obtained on 48 cores on the SCC. On the x-axis we have the heap size as a factor of the minimum heap size under which the programs would run. As we decrease the overall heap size, we see that the programs take longer to run. However, the performance of bi-partitioned heap scheme degrades more gracefully that the single shared heap scheme. At 3X the minimum heap size, bi-partitioned heap is 48% faster than the single shared heap scheme.

In order to study the reason behind this, we separate the mutator time (Figure 6c) and the garbage collection time (Figure 6d). At 3X the minimum heap size, partitioned shared heap collector spends half as much time performing garbage collection when compared to the single shared heap collector. Since our GC involves sequentially scanning the heap, caching even a part of the shared heap improves performance dramatically.

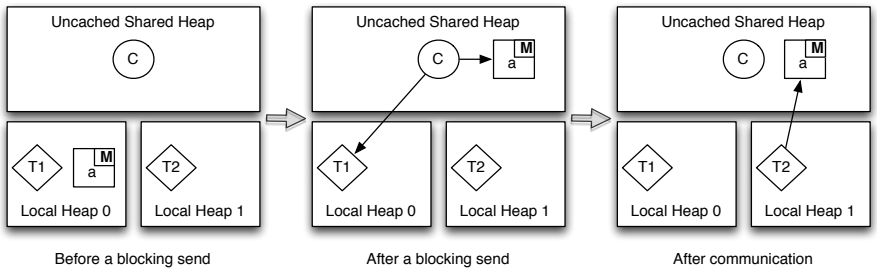


Fig. 4: Steps involved in sending an mutable object `a` by thread `T1` on a shared heap channel `C`, which is eventually received by thread `T2`.

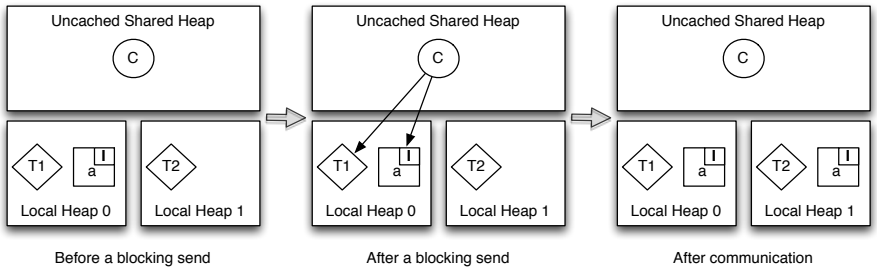
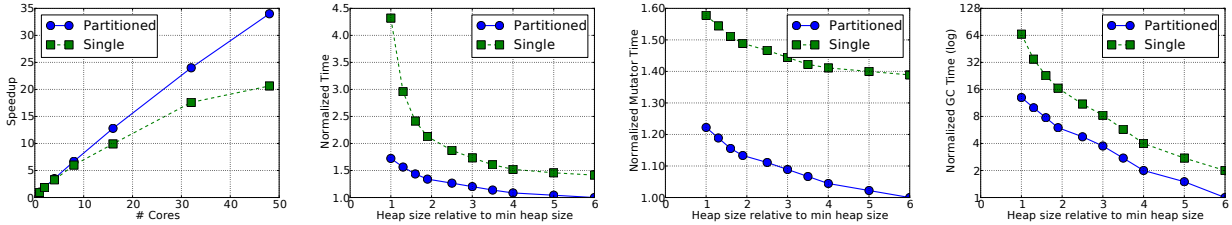


Fig. 5: Steps involved in sending an immutable object `a` by thread `T1` on a shared heap channel `C`, which is eventually received by thread `T2`.



(a) Scalability (b) Total time (48 Cores) (c) Mutator time (48 Cores) (d) GC time (48 Cores)

Fig. 6: Performance comparison of Bi-partitioned shared heap and single, uncached shared heap: Geometric mean for 8 benchmarks.

At 3X the minimum heap size, mutator with the bi-partitioned shared heap is 30% faster than single shared heap implementation. This improvement, although significant, was less than the expected improvement with caching enabled. In order to interpret this improvement, we instrumented our read and write barriers to classify the memory accesses. On average, across all of the benchmarks, 89% of the read or write requests were to the local heap, which is private and is cached both in L1 and L2. This is common to both versions of the local collector. Hence, bi-partitioned shared heap can at best aim to optimize only 11% of the memory requests.

Out of the shared heap memory requests, on average, 93% of all requests were to the cached shared heap. However, it should be noted that cached shared heap data bypass L2, and are only cached in the comparatively smaller L1 cache. Hence, the benefit of caching shared heap data, as far as the mutator is concerned, may not be dramatic if the cached shared heap reads are far and few between. In any case, with bi-partitioned shared heap, less than 1% of mutator accesses were to the uncached memory. Thus, bi-partitioned shared heap local collector is able to potentially cache more than 99% of memory accesses.

B. MPB mapped channels

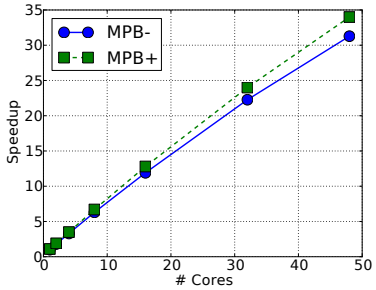


Fig. 7: Performance comparison of first-class channel communication over the MPB `(MBP+)` vs solely over the shared memory `(MPB-)` : Geometric mean over 8 benchmarks.

In order to evaluate the benefit of mapping the first-class

channel communication over the message passing buffer memory, we implemented a version of our communication library that does not use the message passing buffer memory. Recall that if the channel is located in the shared heap, the message in the local heap, and the message does not have a mutable object in its transitive closure, we perform the transfer over the message passing buffer (Case 4 in Section III-C2). Instead, we eagerly globalize the transitive closure of the message and just share the pointer with the receiving thread (similar to Case 3). We call this version MPB-, and the original version MPB+.

Figure 7 shows the performance comparison of MPB+ versus MPB-. On 48-cores, MPB+ is only around 9% faster than the MPB- version. We can attribute several reasons for this marginal improvement. First, we observed that, on average, around only 32% of channel communications were taking advantage of the MPB (Case 4) in the case of MPB+. The rest of the channel communications were either local or were using the shared memory to transfer the messages. Moreover, in the case of MPB-, immutable inter-core messages are transferred over the CSH which is cached.

Second, the cost of inter-core interrupts is substantial, as was observed by others [13], [14]. We measured the time it takes between a core issuing an inter-core interrupt to the time it sends or receives the first byte is around 2000 core cycles. Since majority of the immutable messages exchanged between cores are small, the overhead of setting up the message transfer outweighs the benefit of using the MPB. However, utilizing the MPB prevents immutable messages from being globalized, thus reducing the pressure on the shared memory. As a result, the number of expensive shared heap collections are reduced.

V. Related Work

Software managed cached coherence (SMC) [6] for SCC provides a coherent, shared virtual memory to the programmer. However, the distinction between private and shared memory still exists and it is the responsibility of the programmer to choose data placement. In our system, all data start out as being private, and is only shared with the other cores if necessary. The sharing is performed both through the shared memory as well as over the MPB, based on the nature of message being shared. MESH framework [15] provides a similar mechanism for flexible sharing policies on the SCC as a middle-ware layer.

In the context of mapping first-class channels to MPBs, the work by Prell et al. [16] which presents an implementation of Go's concurrency constructs on the SCC is most similar. However, unlike our channel implementation, channels are implemented directly on the MPB. Since the size of MPB is small, the number of channels that can be concurrently utilized are limited. Moreover, their implementation diverges from Go language specification in that the go-routines running on different cores run under different address spaces. Hence, the result of transferring a mutable object over the channels is undefined.

Our channel communication utilizes both shared memory and the MPBs for inter-core messaging. Barrelfish on the SCC [13] uses MPBs to transfer small messages and bulk transfer is achieved through shared memory. However, Barrelfish differs from our system since it follows a shared-nothing policy for inter-core interaction.

VI. Conclusion

Intel SCC provides an architecture that combines aspects of distributed systems (no cache coherence) with that of a shared memory machine, with support for programmable cache coherence and fast inter-core messaging. In order to effectively utilize this architecture, it is desirable to hide the complexity behind the runtime system.

In this paper, we have presented a port of MultiMLton for the SCC which provides a efficient global address space. Whenever possible and profitable, we utilize MPBs to perform inter-core communication. We observe that while cached shared memory is immensely beneficial, the prohibitive cost of inter-core interrupts diminish the benefits of effectively utilizing the MPBs for first-class channel communication.

Acknowledgments

This work was supported in part by a gift from the Intel Corporation, and National Science Foundation grants CCF-1216613 and CCF-0811631.

References

[1] "SCC Platform Overview," 2012. [Online]. Available: http://communities.intel.com/docs/DOC-5512

[2] R. Milner, M. Tofte, and D. Macqueen, *The Definition of Standard ML*. Cambridge, MA, USA: MIT Press, 1997.

[3] L. Ziarek, K. Sivaramakrishnan, and S. Jagannathan, "Composable Asynchronous Events," in *PLDI*, 2011, pp. 628–639.

[4] J. Reppy, *Concurrent Programming in ML*. Cambridge University Press, 2007.

[5] T. G. Mattson, M. Riepen, T. Lehnig, P. Brett, W. Haas, P. Kennedy, J. Howard, S. Vangal, N. Borkar, G. Ruhl, and S. Dighe, "The 48-core scc processor: the programmer's view," in *SC*, 2010, pp. 1–11.

[6] (2012) Software-Managed Cache Coherence for SCC Revision 1.5. [Online]. Available: http://marcbug.scc-dc.com/svn/repository/tags/SMC_V1.0/smc/SMC.pdf

[7] D. Doligez and X. Leroy, "A Concurrent, Generational Garbage Collector for a Multithreaded Implementation of ML," in *POPL*, 1993, pp. 113–123.

[8] T. A. Anderson, "Optimizations in a Private Nursery-based Garbage Collector," in *ISMM*, 2010, pp. 21–30.

[9] S. Marlow and S. Peyton Jones, "Multicore Garbage Collection with Local Heaps," in *ISMM*, 2011, pp. 21–32.

[10] K. Sivaramakrishnan, L. Ziarek, and S. Jagannathan, "Eliminating read barriers through procrastination and cleanliness," in *ISMM*, 2012, pp. 49–60.

[11] P. M. Sansom, "Dual-Mode Garbage Collection," in *Proceedings of the Workshop on the Parallel Implementation of Functional Languages*, 1991, pp. 283–310.

[12] K. Sivaramakrishnan, L. Ziarek, R. Prasad, and S. Jagannathan, "Lightweight asynchrony using parasitic threads," in *DAMP*, 2010, pp. 63–72.

[13] S. Peter, A. Schüpbach, D. Menzi, and T. Roscoe, "Early experience with the barrelfish os and the single-chip cloud computer," in *MARC Symposium*, 2011, pp. 35–39.

[14] D. Petrović, O. Shahmirzadi, T. Ropars, and A. Schiper, "Asynchronous broadcast on the intel scc using interrupts," in *MARC Symposium*, 2012, pp. 24–29.

[15] Thomas Prescher, Randolf Rotta and Jörg Nolte, "Flexible sharing and replication mechanisms for hybrid memory architectures," in *MARC Symposium*, 2011.

[16] T. R. Andreas Prell, "Gos Concurrency Constructs on the SCC," in *MARC Symposium*, 2012, pp. 2–6.