POSIX-Compliant Portable Code Synthesis for Embedded Systems

Andre Costi Nacul, Siddharth Choudhuri, and Tony Givargis
Department of Computer Science
University of California, Irvine
Center for Embedded Computer Systems
{nacul, sid, givargis}@ics.uci.edu

Technical Report #03-36
November 25, 2003.

Abstract

In a large class of embedded systems, dynamic multithreading using traditional OS techniques is infeasible due to memory and processing overheads or lack of operating systems (OS) availability for the target embedded processor. In this work, we propose a source-to-source translator that takes a POSIX compliant multithreaded C program as input and generates an equivalent, embedded processor independent, single threaded ANSI C program, to be compiled using the embedded processor-specific tool chain. The output of our tool is a highly tuned ANSI C program that embodies the application-specific embedded scheduler and dynamic multithreading infrastructure along with the user code. In this work, we outline the implementation details of our source-to-source translator and show the feasibility of the proposed technique by comparing execution efficiency to approaches based on Java-VM and traditional UNIX based POSIX implementations.

Keywords

Dynamic multithreading, embedded scheduler, code generation, multitasking, scheduling, serializing compilers, software synthesis

Table of Contents

1. Introduction

Embedded software continues to play an ever increasing role in the design of complex embedded applications. In part, the elevated level of abstraction provided by a high-level programming paradigm immensely facilitates a short design cycle, fewer design errors, design portability, and Intellectual Property (IP) reuse. In particular, the concurrent programming paradigm is an ideal model of computation for design of embedded systems, which often encompass inherent concurrency.

On the other hand, embedded systems often have stringent performance requirements (e.g., timing, energy, form-factor, etc.) and, consequently, require a carefully selected and performance tuned embedded processor to meet specified design constraints. In recent years, a plethora of highly customized embedded processors have become available. As an example, Tensilica [1] provides a large family of highly customized application-specific embedded processors (a.k.a., the Xtensa). Likewise, ARM [2] and MIPS [3] provide several derivatives of their respective core processors, in an effort to provide their customers an application-specific solution.

Such embedded processors ship with cross-compilers and the associated tool chain for application development. However, to support a multithreaded application development environment, there is a need for an operating system (OS) layer that can support thread creation, thread synchronization, and thread communication.

Such OS support is seldom available for each and every variant of the base embedded processor. In part, this is due to the lack of system memory and/or sufficient processor performance (e.g., in the case of microcontrollers such as the Microchip PIC [4] and the Phillips 8051 [5]) coupled with the high performance penalty of having a full-fledged OS. Additionally, manually porting and verifying an OS to every embedded processor available is a high-cost job, in terms of time and money.

Thus, there exists a gap in realizing a multithreaded application targeted at a particular embedded processor, as shown in Figure 1.

In this work, we fill this gap by providing a fully automated source-to-source translator that takes a POSIX[1] compliant multithreaded C program as input and generates an *equivalent*, embedded *processor independent*, single threaded ANSI C program, to be compiled using the embedded processor-specific tool chain. The output of our tool is a *highly tuned* ANSI C program that embodies the application-specific embedded scheduler and dynamic multithreading infrastructure along with the user code.

An additional motivation for our work is in the context of hardware/software codesign. Specifically, given a partitioned system, our tool can be used to automatically synthesize the software implementing a cluster of functions $F_1, F_2 \ldots F_N$ that have been mapped to a particular processor P_i.

The remainder of this work is organized as follows. In Section 2, we summarize prior related work. In Section 3, we outline the implementation details of our source to source

---

[1] C extended with POSIX is a standardized way of providing constructs for multi-threaded programming.

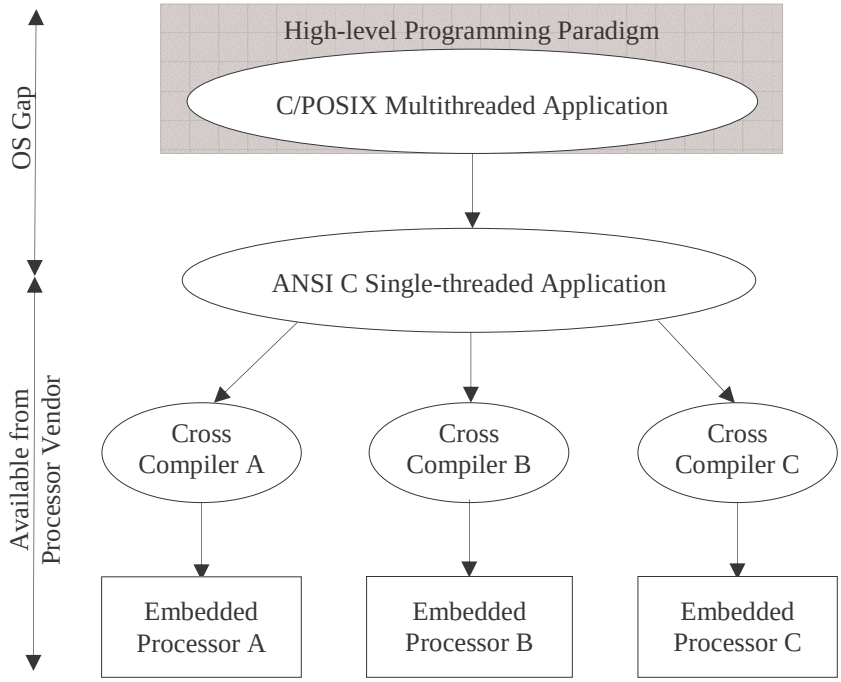


Figure 1: OS gap in embedded processor based designs.

translator. In Section 4, we give our experimental results. Finally, in Section 5, we state our conclusions.

2. Related Work

There are three categories of solutions that partially address the problem stated in this work, namely, a class of virtual machine (VM) based techniques, a class of template based OS generation techniques, and a class of static scheduling techniques.

2.1. VM Based Techniques

In the VM based techniques, an OS providing a multithreaded execution environment is implemented to run on a virtual processor. A compiler for the VM is used to map the application program onto the VM. The virtual processor is in turn executed on the target processor. Portability here is achieved by porting the VM to the desired target embedded processor. The advantages of this class of techniques are that the application and OS code do not require recompilation when moving to a different embedded processor. The disadvantage of this class of techniques is a significant performance penalty (i.e., speed, energy, and memory footprint) incurred by the VM layer, and specifically the VM instruction set interpreter. Moreover, the porting of the VM to the target embedded processor may require more than recompilation efforts. Examples of such VM based techniques are Java [6] and C# [7]. Research in this area tries to address the above-mentioned disadvantages by proposing customized VM for embedded applications [8] or just in time (JIT) compilation techniques [9].

2.2. Template Based Techniques

In the template-based OS generation techniques, a reference OS is used as a template in generating customized derivatives of the OS for particular embedded processors. This class of techniques mainly relies on inclusion or exclusion of OS features depending on application requirements and embedded processor resource availabilities. The disadvantage of this class of techniques is that no single generic OS template can be used in the variety of embedded processors available. Instead, for optimal performance, a rather customized OS template must be made available for each line or family of embedded processor. In addition, for each specific embedded processor within a line or family of processors, an architecture model must be provided to the generator engine.

In one example, Gerstlauer et al. [10] have used the SpecC[2] language, a system-level language, as an input to a refinement tool. The refinement tool partitions the SpecC input into application code and OS partitions. The OS partition is subsequently refined to a final implementation. The mechanism used in this refinement is based on matching needed OS functionality against a library of OS functions.

In a similar approach, Vercauteren et al. [11] have proposed a method based on an API providing OS primitives to the application programmer. This OS template is used to realize the subset of the API that is actually used in the application program.

Finally, Gauthier et al. [12] have proposed an environment for OS generation similar to the previous approaches. Here, a library of OS components that are parameterized is used to synthesize the target OS given a system level description of application program.

2.3. Static Scheduling Techniques

In the static scheduling based techniques, it is assumed that the application program consists of a static and a priori know set of tasks. Given this assumption, it is possible to compute a static execution schedule, in other words, an interleaved execution order and generate an equivalent monolithic program. The advantage of this class of approaches is that the generated program is application-specific and thus highly efficient. The disadvantage of this class of techniques is that dynamic multithreading is not possible. Our technique specifically addresses the dynamic multithreading issue. Moreover, our technique is orthogonal to such static scheduling. For example, the set of a priori know static threads can be scheduled using static scheduling while the dynamically created threads can be handled by a technique similar to ours. A very good general survey on generating sequential code for a static set of tasks is done by Edwards [13].

In a more specific example, Lin [14] has proposed a technique that takes as input an extended C code that includes primitives for inter-task communication based on channels, as well as primitives for specifying threads and generates ANSI C code. The mechanism here is to model the static set of tasks using a Petri Net and generate a single threaded code simulating a correct execution order of the Petri Net. Similar techniques have also been proposed by Cortadella et al. [15][16].

---

[2] The multitasking allowed in SpecC is limited to a static and a priori known set of concurrent tasks.

3. Technical Approach

3.1. Introduction

Input to our translator is a multithreaded program P_{input}, written in C extended with POSIX [17]. The basic constructs provided by POSIX are functions for task creation and management (e.g., `pthread_create`, `pthread_join`, `pthread_cancel`, etc.) as well as a set of synchronization variables (e.g., `sema_t`, `mutex_t`, etc.). Output of our system is a single-threaded strict ANSI C program P_{output} that is equivalent in function to P_{input}. More specifically, P_{output} does not require any OS support and can be compiled by any valid ANSI C compiler into a self sufficient binary for a target embedded processor.

To support multithreading there is a need for efficient sharing of the processor among multiple threads, providing synchronization mechanisms, and communication primitives. Sharing of the processor among threads requires preemption and, in turn, preemption requires a mechanism for saving/restoring thread specific information (i.e., the task context). In conventional approaches, multithreading is implemented within the OS. When a thread T_i is created, OS allocates sufficient memory for saving T_i's context information (e.g., registers, function call stack, program counter, etc.). Periodically, an interrupt generated by the system timer invokes the OS scheduler. The scheduler saves the context of the currently executing task T_{old} and restores the context of a new task T_{new} to be executed. The OS, in turn, relies on the underlying processor for invoking the scheduler (i.e., via a timer interrupt), context switching (register load/store instructions), and synchronization (i.e., test-and-set instruction).

In our approach, the challenge is to achieve the same at a higher level of abstraction, namely, by using the mechanisms provided by strict ANSI C language. In the next section, we give our implementation details for source-level multithreading.

3.2. Preemption and Scheduling

In our implementation, the basic unit of execution, scheduled by the scheduler, is called an *atomic execution block* (AEB). An AEB is a block of code that is executed in its entirety prior to scheduling the next AEB. A thread T_i is partitioned into an AEB graph whose nodes are AEBs and edges represent control flow. For example, in Figure 2 the AEB graph of thread *gcd* is composed of nodes N_0, N_1, N_2, and N_3. Within an AEB graph, each node is implemented as an ANSI C function whose return value encodes a pointer to the next node in that graph. For example in Figure 2, lines 32-58, the function `n1` corresponds to node N_1 and returns a pointer to `n2` or `n3`, representing nodes N_2 or N_3 depending on the runtime behavior of the program. We note that in our implementation, the partitioning is performed on the basic block intermediate representation of the input source program. Moreover, we note that an AEB node may be composed of one or more basic blocks. We return to the topic of partitioning and its implications on the runtime behavior in a later section.

During runtime, we maintain the following context information for each thread that has been created: memory to store the intermediate variables computed by a partially executed thread called *live* (this is accomplished by performing a live variable analysis during the partitioning of the program into AEB nodes) and a pointer to the next AEB

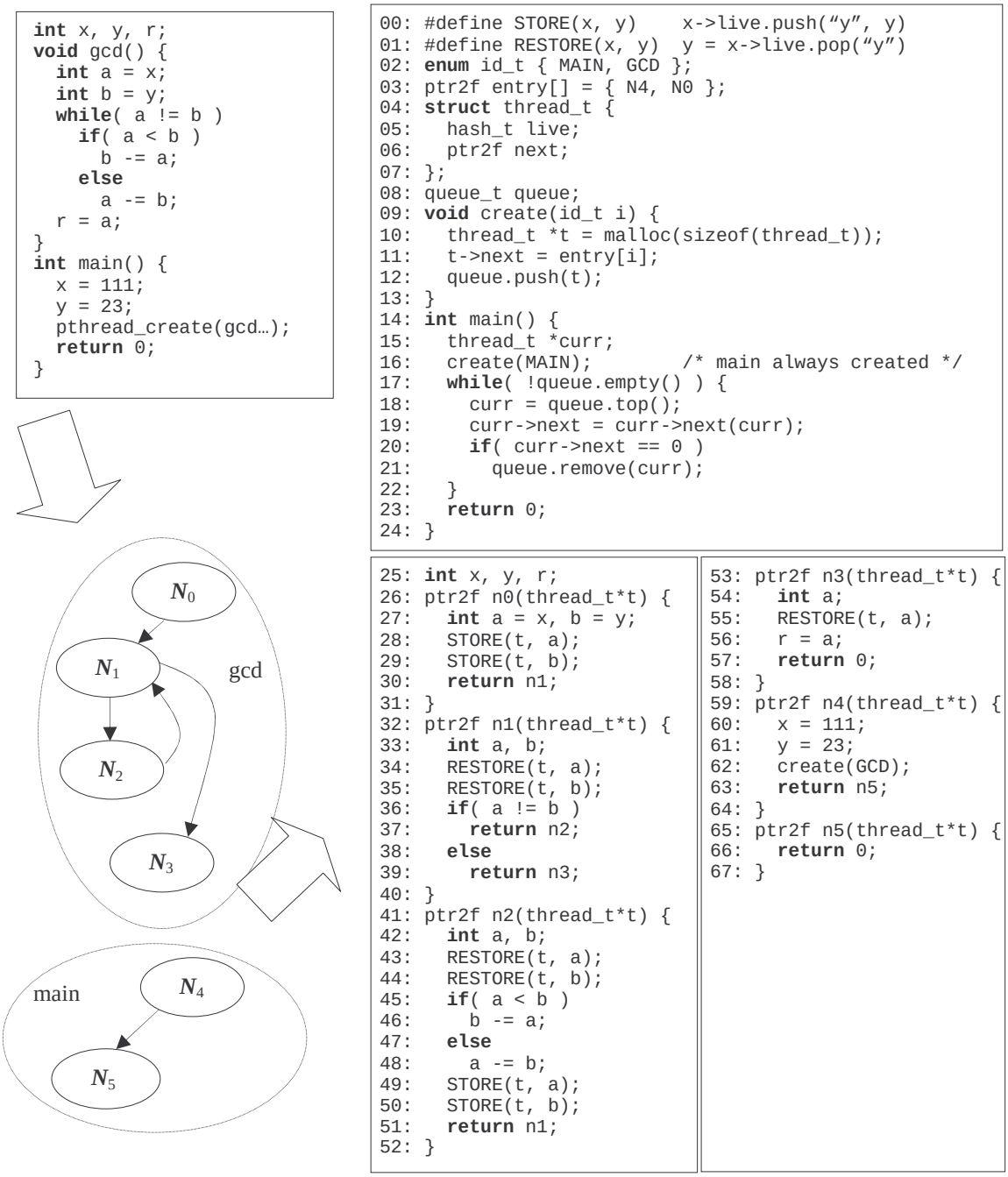
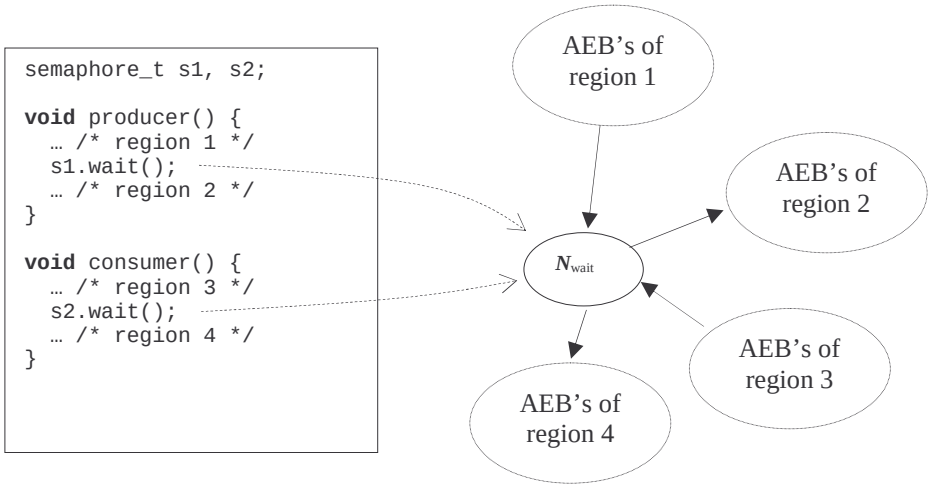


Figure 2: Source-level multithreading.

node that is to be executed some time in the future, called *next*, as shown in Figure 2, lines 4-7.

When a thread is created, the above context for it is allocated, the next field is initialized to the entry AEB of the thread, and the thread context is pushed onto a queue of existing



(a)

```
00: struct semaphore_t {
01:   int value;
02:   queue_t waiting;
03: };
04: ptr2f wait(thread_t *t, semaphore_t *s) {
05:   if( s->value == 0 ) {
06:     s->waiting.push(t);
07:     t->status = BLOCKED;
08:     return t->curr;
09:   }
10:   value--;
11:   return t->next
12: }
13: ptr2f signal(thread_t *t, semaphore_t *s) {
14:   s->value++;
15:   while( !s->waiting.empty() )
16:     (s->waiting.pop())->status = RUNNABLE;
17:   return t->next;
18: }
```

(b)

Figure 3: Semaphore synchronization primitive implementation.

threads, called *queue*, to be processed by the embedded scheduler, as shown in Figure 2, lines 8-13.

The embedded scheduler is responsible for selecting and executing the next thread as shown in Figure 2, lines 14-22. The embedded scheduler always creates the main thread, corresponding to the `main` function of the input C program. Then, as long as the queue of existing threads is not empty, the scheduler selects the thread with the highest priority T_i, or in the case of a priority tie, the one with the next highest identifier. The next AEB pointer of T_i, pointing to f_i, is used to resume the execution of T_i by making a function call to f_i. The return value of f_i is in turn used to update the next AEB of T_i. A return value of zero indicates that T_i has reached its termination point, and thus is removed from the queue of existing threads. The scheduling algorithm here is a priority based scheme, as

defined by the POSIX. The way priorities are assigned to threads, as they are created, can enforce alternate scheduling schemes, such as round-robin, in the case of all threads having equal priority, or earliest deadline first (EDF), in the case of threads having priority equal to the inverse of their deadline, priority inversion, and so on.

3.3. Synchronization

We implement the basic *semaphore* (`sema_t` in POSIX) synchronization primitive, upon which any other synchronization construct can be built. A semaphore is an integer variable with two operations, *wait* and *signal* (`sema_wait` and `sema_post` in POSIX). A thread T_i calling wait on a semaphore S will be blocked if the S's integer value is zero. Otherwise, S's integer value is decremented and T_i is allowed to continue. T_i calling signal on S will increment S's integer value and unblock any thread that is currently blocked waiting on S.

To implement semaphores, we add to a thread Ti's context two additional fields called status and current. Status is one of blocked or runnable and is set appropriately when a thread is blocked waiting on a semaphore. The current field of a thread is similar to the next field (see Figure 2, line 6) but at any given time points to the current AEB that is being executed. A thread is always partitioned into AEBs when semaphore wait or signal operations are encountered. In other words, the semaphore wait and signal primitives always reside in their own AEBs, as shown in Figure 3(a). Moreover, these special AEBs are shared nodes connecting AEB graphs of multiple threads.

We implement a semaphore as a data structure with an integer field and a queue of waiting threads, as shown in Figure 3(b), lines 0-3. A wait operation on a semaphore S checks the value of S and, if zero, blocks the calling thread T_i by setting T_i's status to blocked, adds T_i to S's queue of waiting threads, and returns control to the embedded scheduler. However, instead of returning a pointer to the next AEB of T_i, it returns a pointer to the current AEB of T_i (i.e., the one containing the semaphore wait or signal call) so that when T_i is unblocked, S's variable is rechecked. If S's value is nonzero, it is decremented and control is returned to the embedded scheduler with a pointer to the next AEB of T_i, following the wait operation. A signal operation on S increments the value of S, unblocks all the threads in the waiting queue of S, and returns control to the embedded scheduler with a pointer to the next AEB of T_i, following the signal operation.

3.4. Partitioning

As described earlier, the partitioning of the code into AEB graphs is the key to implementing multithreading at a high-level of abstraction. Recall that boundaries of AEB represent the points where threads might be preempted or resumed for execution. Some partitions are unavoidable and must be performed for correctness, specifically, when a thread invokes a synchronization operation, or when a thread creates another thread. In the case when a thread invokes a synchronization operation and thus is blocked, the embedded scheduler must regain and transfer control to one of runnable threads. Likewise, when a thread creates another, possibly higher priority, thread, the embedded scheduler must regain and possibly transfer control to the new thread in accordance with the priority based scheduling technique. However, partitioning beyond what is needed for correctness, impacts timing issues as described next.

In general, partitioning will determine the granularity level of the scheduling (i.e., the time quantum), as well as the thread latency. A good partitioning of the threads into AEBs would be one where all AEBs have approximately the same average case execution time μ and a relatively low deviation λ from the average, which can be computed if the average case execution time of each AEB is known. Note that the average case execution time W_i of an AEB N_i is defined as the time taken to execute the code C_i in N_i plus the time taken to store and restore all live variables V_i at the entry and exit of N_i. Moreover, an estimate of V_i can be obtained by performing a live variable analysis. An estimate of C_i can be obtained by static profiling. In an iterative approach, our partitioning heuristic refines an existing partition and evaluates the average case execution times until an acceptable partition is discovered.

3.5. Interrupts

Preempting an AEB when an interrupt occurs would break the principle that every AEB executes until completion without preemption. Instead, the code for an interrupt service routine *I* is treated as a thread. On an interrupt destined for *I*, a corresponding thread is created, having a priority higher than all existing threads. Note that if multiple interrupts destined for *I* occur, multiple threads will be created and scheduled for execution. This is a uniform and powerful mechanism for handling interrupts in a multithreaded environment. However, the latency for handling the interrupt will depend on the average execution time of the AEBs, which in turn depends on the partitioning scheme used, as described in the previous section.

4. Experiments

Our experimental flow is depicted in Figure 4. The multithreaded C extended with POSIX application is compiled with a generic front-end compiler to obtain the basic block (BB) control flow graph (CFG) representation. This intermediate representation along with an annotated (i.e., identified POSIX primitives) version of the input source application, is used by the partitioning module to generate the AEB graphs. Then, a live variable analysis is performed on the AEB graphs and the result is fed back to the partitioning module to refine the partitions until acceptable preemption timing and latency is achieved. The resulting AEB graphs are then passed to the code generator to output the corresponding ANSI C functions for each AEB node. In addition, the embedded scheduler along with other C data structures and synchronization APIs is included from the generic C structures library.

The above experimental flow has been successfully applied to two classical concurrent problems, the *consumer-producer* problem and the *dining philosophers* problem. We have compared the performance of the generated output with two other implementations of the same problems. One of the implementations has been done in C with POSIX threads, using the Solaris POSIX libraries (i.e., an OS based approach), and the other has been implemented in Java (i.e., a VM based approach). Our results are presented in Figure 5.

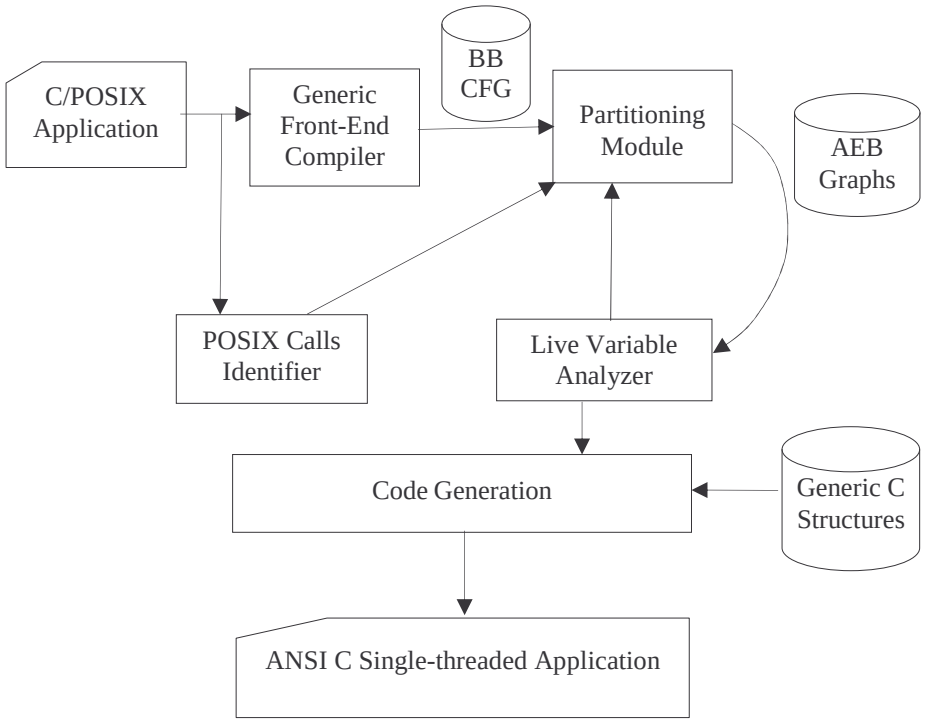


Figure 4: Experimental setup.

In our results, the OS based approach has a slightly better performance, in terms of execution time, due to fewer context switches, given the default time quantum of the underlying preemption scheme. As expected, the VM based approach has worse

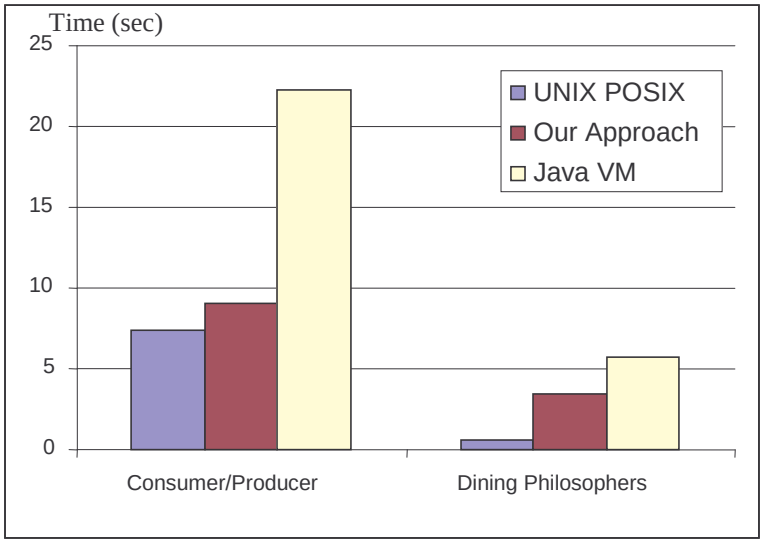


Figure 5: Experimental results.

performance in terms of execution time due to the overhead of interpreted VM code.

Among all experiments, the size of the binary executable of our scheme is 35KB (consumer-producer) and 37KB (dining philosophers), the size of the binary executable of the OS based scheme is 24KB (consumer-producer) and 26KB (dining philosophers), and the size of the byte code in the VM based scheme is 6KB (consumer-producer) and 3KB (dining philosophers). Moreover, we point out that our binary executable is a self sufficient executable which does not require any additional API or OS support. In the case of the OS based scheme, the OS footprint must also be considered. Likewise, in the case of the VM based technique, the VM layer footprint must also be considered. We note that binaries were generated by using *gcc* version 3.2 or *javac* version 1.4.1 for the Sun Solaris OS version 8.0.

5. Conclusions

We have presented a scheme for source-to-source translation of a multithreaded application written in C extended with POSIX into a single-threaded ANSI C program which can be compiled using a standard C cross-compiler for any target embedded processor. While compiler tool chains are commonly available for any of the large number of customized embedded processors, the same is not true for operating systems, which traditionally provides the primitives for multithreading at the application level. Our source-to-source translator fills this missing OS gap by automatically generating a platform independent C program that encapsulates multithreading support customized for the input application.

Our future direction is to explore more efficient source-to-source translation schemes that can take advantage of abstract architecture description models and application profiling information. Specifically, our partitioning algorithm can benefit from knowledge of the underlying target embedded processor combined with profiling information in better estimating the execution time of an AEB as depicted in Figure 6. In addition, extension

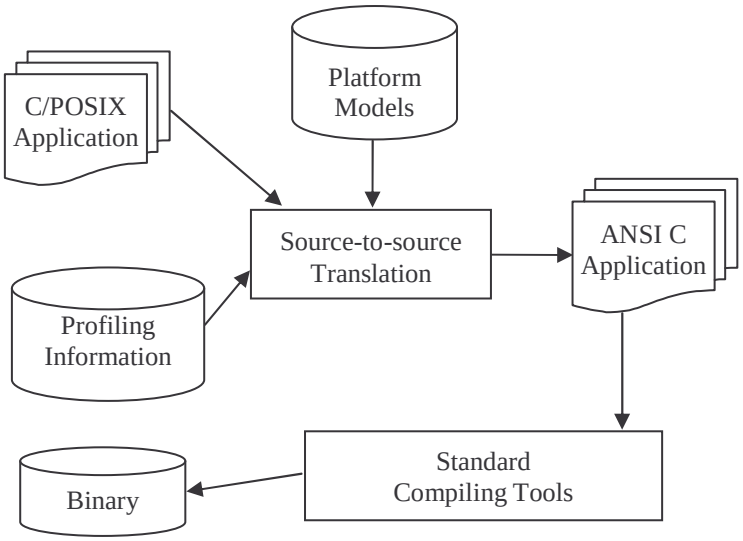


Figure 6: Future design flow.

for soft and firm real-time systems can also be implemented which attempt at providing more stringent and deterministic timing behavior.

6. Acknowledgments

This work was supported in part by a National Science Foundation Award (#0205712) and by a CAPES Foundation, Brazil scholarship (#1054015).

7. References

[1] Tensilica Inc. www.tensilica.com.

[2] ARM Inc. www.arm.com.

[3] MIPS Inc. www.mips.com.

[4] Microchip Inc. www.microchip.com.

[5] Philips Inc. www.philips.com.

[6] J Gosling, B. Joy, G. Steele. The Java Language Specification. Addison-Wesley, 1996.

[7] Microsoft Corporation. The C# 2.0 Specification. Available at http://msdn.microsoft.com/vcsharp. July 2003.

[8] V.C. de Verdiere, S. Cros, C. Fabre, R. Guider, S. Yovine. Speedup Prediction for Selective Compilation of Embedded Java Programs. Proceedings of EMSOFT, October 2002.

[9] J. Aycock. A Brief History of Just-In-Time. In ACM Computing Surveys, v. 35, n. 2, June 2003. pp 97-113.

[10] Gerstlauer, H. Yu, D. Gajski. RTOS Modeling for System Level Design. Proceedings of DATE, March 2003.

[11] S. Vercauteren, B. Lin, H. De Man. A Strategy for Real-Time Kernel Support in Application-Specific HW/SW Embedded Architectures. Proceedings of DAC, 1996.

[12] L. Gauthier, S. Yoo, A. Jerraya. Automatic Generation and Targeting of Application-Specific Operating Systems and Embedded Systems Software. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems. v. 20, n. 11, November 2001, pp 1293-1301.

[13] S. Edwards. Tutorial: Compiling Concurrent Languages for Sequential Processors. ACM Transactions on Design Automation of Electronic Systems, v.8, n.2, April 2003, pp 141-187.

[14] B. Lin. Efficient Compilation of Process-Based Concurrent Programs without Run-Time Scheduling. Proceedings of DATE, February 1998.

[15] J. Cortadella et. al. Task Generation and Compile-Time Scheduling for Mixed Data-Control Embedded Software. Proceedings of DAC 2000.

[16] J. Cortadella et. al. Quasi-static scheduling of independent tasks for reactive systems. Lecture Notes in Computer Science, June 2002.

[17] POSIX Open Group. http://www.opengroup.org.