Modeling and Verifying Abstract Multithreaded Systems

Winfried Grünewald and Klaus Schneider

Institut für Rechnerentwurf und Fehlertoleranz (Prof. Dr. Schmid)
Universität Karlsruhe
Postfach 6980, 76128 Karlsruhe, Deutschland
email: {Winfried.Gruenewald, Klaus.Schneider} @informatik.uni-karlsruhe.de
http://goethe.ira.uka.de

Abstract. Multithreaded systems consist of programs (threads), an operating system and one or more processors. We describe abstraction techniques from programs, the operating system and processors to extract a finite state machine that reflects the thread manager in its system context. To assure the correctness and fairness of the concurrent execution, we apply common verification techniques, in particular CTL model checking and theorem proving in higher order logic. Additionally, requirements for interrupt service routines are derived. We illustrate the application of the method by verifying the thread management of the multithreaded Rhamma system.

1 Introduction

We consider embedded systems built of software and hardware, where the software is divided into several interacting threads. The number of processors in embedded systems is usually far less than the number of threads such that a thread management to schedule the threads on the available resources is required. These thread managers are normally implemented by an operating system kernel to provide the programmer with a virtual thread model that has an unlimited number of threads. Using virtual thread models, the programmer does not need to consider the available resources. Clearly, the correctness and fairness of the thread manager is essential for the correctness of the entire system. In particular, the thread manager must assure that each thread has after some time access to all resources required for its execution.

In this paper, we present a method for modeling the thread management of embedded systems in an abstract manner and show that this allows to apply verification methods to assure the correctness of the concurrent execution. The abstraction retains all details of the parts of the programs, the operating system and the hardware which affect the thread handling, while all other parts are given as abstract as possible. Our basic aim is to show that the virtual thread model is correctly implemented by the system.

For this reason, we consider programs as sequences of instructions that are directly executable on the processors. However, we do only consider those instructions that affect the thread handling, while others are replaced by ‘noop’ instructions that have no effect apart from consuming processor cycle time. For example, the remaining instructions can generate new threads, start or stop existing threads. Note however, that most thread managers are not able to deal with unreasonable programs that e.g. try to stop non-existing threads. For this reason, we must exclude these programs, and assume that the programmer takes care that these unreasonable cases do not occur.

1This work has been partly financed by DFG project ‘Verification of Embedded Systems’ (see http://goethe.ira.uka.de/ for more information).
On the other hand, our abstraction of the instruction set allows us to cover all reasonable programs, instead of a given concrete one. Hence, our aim is to show that the thread manager itself is correct in the sense that it work for each reasonable set of threads. The definition of reasonable programs depends on the considered system and we explain it in detail for the Rhamma architecture [6]. The abstraction of the instructions in the programs is also our abstraction of the instruction set architecture of the microprocessor.

We do not distinguish in our model between hardware and the underlying operating system. As far as thread management is concerned, the partition into software (operating system) and hardware is to a large extent variable. Both parts interact with each other by means of interrupt service routines that are usually implemented in software, i.e. in the operating system (to make the system flexible). In order to relate the virtual thread model with the underlying implementation, we do also abstract away from the implementation of these interrupt service routines and do only derive the required effects they must have on the thread management.

If we additionally assume that there is an upper bound for the number of currently existing threads, then the presented abstraction techniques reduce the behavior of the thread manager to a finite-state machine. For this reason, common symbolic traversal techniques [3] can be used for its verification.

There are two main properties that must hold for the correctness of the thread management: on the one hand, it must be shown that the thread manager implements the virtual thread model. On the other hand, the fairness of the thread manager is to be verified. This involves to prove that each thread will be executed after some time and has access to all resources it requires for its execution.

To illustrate the application of our verification method, we have verified the thread manager of the Rhamma microprocessor architecture [6, 7]. Rhamma is a block-multithreaded processor that uses an extremely fast context switch to bridge latencies arising e.g. of memory access. Rhamma uses a build-in thread manager implemented in hardware. Due to the flexibility of the thread manager, it must interact with the operating system [2] via interrupt routines. These routines are responsible for generating new thread tags, writing out and reading in register frames.

The outline of the paper is as follows: in the next section, the virtual thread model (VTM), the memory model as seen by the operating system (OSM) and the model of the microprocessors (MPM) are presented in detail. Section 3 discusses the specifications we have verified and some experimental results. The paper ends with some final conclusions. In the following, paragraphs containing lots of details necessary for a deeper understanding are written in small font.

2 Modeling Rhamma’s Thread Manager

As already mentioned, Rhamma is a multithreaded microprocessor architecture with a thread management component implemented in hardware. Rhamma has a fixed number $m$ of physically existing register sets for handling $m$ different threads on the processor. However, there is only one execution pipeline, hence the threads on the processor are executed in an interleaving technique. Nevertheless, the use of several register sets allows extremely fast context switches from one thread to another. Rhamma’s instruction set provides directly instructions for generating threads (getfr), releasing a thread $j$ (relfr($j$)), starting thread a $j$ (start($j$)), and stopping

\[2\text{Therefore, the interrupt service routines might also be given in hardware in our model.}\]

\[3\text{Of course such an upper bound will always exist for each reasonable system, otherwise it could not be implemented with finite hardware. Note also that this does allow to generate arbitrary many threads over the time as long as enough threads do terminate such that the upper bound holds.}\]
the current thread (stop). All other instructions are replaced due to our abstraction by the \textit{noop} instruction.

\section*{2.1 Virtual Thread Model (VTM)}

In this section, the virtual thread model (VTM) of an embedded system based on Rhamma is described. The VTM uses thread frames as containers to provide some management information about the thread. A thread can generate a new thread by looking for a free thread frame, loading a program counter into that frame and starting the program. If the thread terminates, another program can be loaded to recycle its thread frame afterwards.

The VTM can control an unlimited number of thread frames $T_{VTM}$. Each thread frame $T_{VTM}$ is uniquely identified by its thread tag $i \in \mathbb{N}$ and has a state $S_{VTM} \in \{\text{free}, \text{stopped}, \text{active}\}$ as well as a current instruction $I_{VTM} \in \{\text{getfr, relfr}(j), \text{start}(j), \text{stop}, \text{noop}\}$. The state transition diagram of a thread frame $T_{VTM}$ in the VTM is given in figure 1. The frame remains in its current state unless one of the signals is present.

\begin{figure}[h]
\centering
\begin{tikzpicture}[node distance=2cm, thick]
  \node (free) {$S_{VTM} = \text{free}$};
  \node (stopped) [right of=free, xshift=3cm] {$S_{VTM} = \text{stopped}$};
  \node (active) [below of=stopped] {$S_{VTM} = \text{active}$};
  \draw[->] (free) -- (stopped) node [midway, above] {getfr$_i$};
  \draw[->] (stopped) -- (active) node [midway, above] {reldr$_i$};
  \draw[->] (active) -- (free) node [midway, above] {reldr$_i$};
  \draw[->] (active) -- (stopped) node [midway, above] {stop$_i$};
  \draw[->] (stopped) -- (active) node [midway, above] {start$_i$};
\end{tikzpicture}
\caption{State transitions of thread $T_{VTM}^i$ in the VTM}
\end{figure}

In state free, the thread frame is not yet associated with a program. In state stopped a program is already associated with the frame, however it is still not executed. Finally, the program will be executed if the frame is in state active. A program error occurs if the frame releases another thread that is not in state stopped. Also, a thread to be started have to be in state stopped. All programs with instruction sequences that do not have these two errors are viewed in Rhamma as \textit{reasonable}.

The semantics of the instructions and the signals in figure 1 are as follows:

- $I_{VTM} = \text{getfr}$ generates a new thread by choosing a thread frame $T_{VTM}$ in state free and emits the signal getfr$_j$. The frame $j$ is associated with a program and switches to state stopped.
- $I_{VTM} = \text{reldr}(j)$ releases frame $T_{VTM}$ and emits the signal relfr$_j$. The frame $j$ switches to state free and the association with its program is broken.
- $I_{VTM} = \text{start}(j)$ starts thread $j$ and emits the signal start$_j$. The frame $j$ switches to state active.
- $I_{VTM} = \text{stop}$ stops the current thread and emits the signal stop$_j$. The next state of $T_{VTM}$ is stopped. Another thread can restart or release the thread.
- $I_{VTM} = \text{noop}$ is an instruction that does not affect the thread frames and emits no signals. The processor executes an arithmetic-logic, a branch or a memory access instruction.

\section*{2.2 Model of the Thread Manager}

Usually, the number of existing threads will be larger than the number of available register sets on the processor. Hence, some of the threads must be stored in the memory until they can
proceed further. Threads that currently have no register set on the processor are managed by the operating system while the other threads are managed directly by the processor. For this reason, the model of the thread management is split into a part modeling the microprocessor (MPM) and the operating system (OSM). Both parts interact with each other by means of interrupt service routines.

### 2.2.1 Model of the Operating System (OSM)

Similar to the VTM, an unlimited number\(^4\) of thread frames \(T_{i}^{OSM}\) is available in the operating systems model (OSM). A thread frame \(T_{i}^{OSM}\) is identified with its unique tag \(i \in \mathbb{N}\) and has a state \(S_{i}^{OSM} \in \{\text{free}, \text{stopped}, \text{active}, \text{on\_processor}\}\). As both the OSM and the MPM implement the VTM, there must be a correspondence between the various states. The OSM can only distinguish between the states of the thread when it is in the memory. In this case, \(S_{i}^{OSM} = S_{i}^{VTM}\) holds, otherwise \(S_{i}^{OSM} = \text{on\_processor}\) holds which means that the OSM has no further information on the virtual thread state.

![Figure 2: Behavior of thread frame \(i\) in the operating systems model](image)

While in the VTM each running thread frame executes a program and can hence generate signals for the thread handling, the frames in the OSM are passive and are only triggered by the processor. Changing a threads state in the OSM is only possible when the thread is first loaded to the processor and uploaded afterwards. Loading threads on and uploading threads from the processor is done by the interrupt service routines as given in figure 2. Depending on the state of the thread, different routines are used: \texttt{writeOut} and \texttt{readIn} for threads in state \textit{stopped}, \texttt{o\_load} and \texttt{u\_load} for threads in state \textit{active}, and \texttt{freeTag} and \texttt{newTag} for threads in state \textit{free}.

### 2.2.2 Model of the Microprocessor (MPM)

The Rhamma microprocessor has \(m\) register sets which are managed by register frames \(F_{1}, \ldots, F_{m}\). Thus, Rhamma is able to directly manage \(m\) threads by a on-chip controller without using the operating system to some extent. As this part of Rhamma must also implement in some way the VTM, each register frame has in principle the states of the VTM. As the interaction between the MPM and the OSM is done via interrupt service routines, we must further introduce intermediate states that refine the VTM states by the state of the interrupt procedure. Also, it might be the case that there are register frames that have not been associated with a thread so far, such that an additional state \texttt{rs\_free} with its refinements is required.

Each \(F_{j}\) has a state \(S_{j}^{\mu}\), a thread tag \(\tau_{j}\), possibly a program with a current instruction \(I_{j}^{\mu}\), possibly the tag of a partner frame \(p_{j}\), and of course the registers of the thread. \(\tau_{j} = i\) means

---

\(^4\) We assume that the main memory is large enough.
Figure 3: State transition of a register frame $i$ on the processor
The execution depends on some cases:

\[ I_{i}^{\mu} = \text{getfr} \]

The execution depends on some cases:

\[ \exists j, S_{j}^{\mu} = \text{free}_0: \text{a frame } F_j \text{ in state } S_{j}^{\mu} = \text{free}_0 \text{ is selected and the signal } \text{getfr}_j \text{ is emitted. Hence, } F_j \text{ switches to state stopped}_i. \]

\[ \exists j, S_{j}^{\mu} = \text{rsfree}_0: \text{a frame } F_j \text{ in state } S_{j}^{\mu} = \text{rsfree}_0 \text{ is chosen and the signal } \text{getfr}_j \text{ is emitted such that } F_j \text{ switches to state } \text{rsfree}_{i}^{\text{newTag}}. \text{The partners are set to } p_i := F_j \text{ and } p_j := F_i \text{ and } F_i \text{ switches to active}_i. \]

If \( F_j \) is enabled by \( e_j \) later on, \( F_j \) executes the routine \( \text{newTag} \) and switches to state \( \text{free}_{i}^{\text{newTag}}. \) \( \text{newTag} \) chooses a thread frame \( T_{\tau_j}^{{\text{OSM}}} \) that is in state free and assigns \( \tau_j := \tau. \) \( T_{\tau_j}^{{\text{OSM}}} \) will switch to state on processor. The termination of \( \text{newTag} \) generates the signal restart, such that the next state of \( F_i \) is active\(_{i}^{\text{restart}}. \)

In state active\(_{i}^{\text{restart}} \), \( F_i \) again executes the instruction getfr. In this state, the partner is determined and the signal getfr\(_j \) is repeated. The next states of \( F_i \) and \( F_j \) are hence active\(_i \) and stopped\(_i \), respectively.

\[ \exists j, S_{j}^{\mu} = \text{stopped}_i: \text{The processor chooses a frame } F_j \text{ in state } \text{stopped}_i, \text{ generates the signal } \text{getfr}_j \text{ and sets the partners } p_i := F_j \text{ and } p_j := F_i. \text{ The next states of } F_i \text{ and } F_j \text{ are active}_i \text{ and stopped}_{i}^{\text{reload}}, \text{ respectively.} \]

When \( F_j \) is enabled by \( e_j \), writeOut is executed by \( F_j \) and \( F_j \) switches to rsfree\(_i^{\text{newTag}}. \) writeOut loads up the thread \( \tau_j. \) \( T_{\tau_j}^{{\text{OSM}}} \) switches to state stopped. The situation \( S_{i}^{\mu} = \text{active}_i, S_{j}^{\mu} = \text{rsfree}_{j}^{\text{newTag}} \) is handled as in the previous case.

**Otherwise:** All frames are either active or involved in an interrupt. If none of the frames are in state active\(_0^{\text{waiting}} \), the signal \( \text{a}_i \) is emitted and \( F_i \) switches to active\(_0^{\text{waiting}} \), otherwise oload is emitted and \( F_j \) switches to active\(_j^{\text{reload}}. \) The next enable signal will start the routine oload to upload thread \( \tau_j \) from the processor. \( F_i \) switches to state rsfree\(_0. \)

\[ I_{i}^{\mu} = \text{relfr}(\tau) \]

\( \tau_{i} = \tau \) The signal relfr\(_i \) is emitted and \( F_i \) switches to state free\(_i \) (the thread terminates).

\[ \exists j, \tau_{j} = \tau \] The signal relfr\(_j \) is emitted and \( F_j \) switches to state free\(_0 \). Note that \( F_j \) is uniquely determined and \( S_{j}^{\mu} = \text{stopped}_i \) must hold.

\[ \exists j, S_{j}^{\mu} = \text{rsfree}_0: \text{a frame } F_j \text{ in state } \text{rsfree}_0 \text{ is chosen, relfr}_j \text{ is emitted, and the partners are set.} \]

The next states of \( F_i \) and \( F_j \) are active\(_i \) and free\(_i^{\text{reload}} \), respectively and the thread tag\( \tau_j \) is assigned to\( \tau. \)

When \( F_j \) is enabled by \( e_j \), the routine readIn is executed by \( F_j \) and \( F_j \) switches to stopped\(_i^{\text{readIn}}. \) readIn loads the thread \( \tau \) (\( T_{\tau_j}^{{\text{OSM}}} \) switches to state on processor). Termination of readIn emits restart\(_j \) such that \( F_i \) switches to active\(_0^{\text{restart}}. \)

\( F_i \) again executes relfr\(_i(\tau). \) As the partner is determined, relfr\(_j \) is reemitted. The next states of \( F_i \) and \( F_j \) are active\(_i \) and free\(_0 \), respectively.
This case is similar to relfr: a frame $F_j$ in state free is chosen, relfr is emitted, and the partners are set. The next states of $F_i$ and $F_j$ are active and free, respectively.

When $F_j$ is enabled by $e_j$, freeTag is executed and $F_j$ switches to $rsfree_1$. freeTag empties the thread frame $T_{r_j}$ ($T_{r_j}^{OSM}$ will switch to state free). The thread tag $T_{r_j}$ is invalidated in frame $F_j$, but the partner is stored, of course. The termination of freeTag emits restart such that the next state of $F_i$ is active restart.

In state active restart, $F_i$ again executes relfr($\tau$) and relfr is reemitted. The next states of $F_i$ and $F_j$ are active and free, respectively. The situation $S_i^{\mu} = active_1, S_j^{\mu} = rsfree_1$ appeared already above.

$\exists j, S_j^{\mu} = stopped_1$: a frame $F_j$ in state stopped is chosen, relfr is emitted, and the partners are set. The next states of $F_i$ and $F_j$ are active and stopped, respectively.

When $F_j$ is enabled by $e_j$, $F_j$ executes writeOut and switches to $rsfree_1$. The termination of writeOut emits restart and $F_i$ switches to $active_1$ restart. The situation $S_i^{\mu} = active_0$, $S_j^{\mu} = rsfree_1$ appeared already above.

Otherwise: All frames are either active or involved in an interrupt. This case is handled as the ‘otherwise’ case of instruction getfr.

$T_i^{\mu} = start(\tau)$ This case is similar to relfr($\tau$). However, the case $\tau_i = \tau$ can not occur, as an already running thread can not start itself. All other cases are handled as the corresponding cases for the instruction relfr($\tau$), except for the last transition of $F_j$ that ends in each case in state active, instead of free.

$T_i^{\mu} = stop$ suspends the execution of the thread $\tau_i$. $F_i$ switches to state stopped where it can be restarted by another frame.

$T_i^{\mu} = noop$ emit no signal and does hence not lead to any transition.

To assure fairness, the thread manager uses a timeout mechanism. In our model, we abstract away from the implementation the timeout mechanism, rather we only demand that for each $F_i$, the signal timeout is infinitely often emitted together with $e_i$. This will cause the transition from states active, active restart to active load. The next enable signal for $F_i$ calls the routine oload and $F_i$ switches to $rsfree_0$.

When all register frames are in state $rsfree$, active or stopped, while there is a thread in the OSM in state $S_i^{OSM} = active$, Rhamma has the ability to emit a signal $uload$. Here the following cases can be distinguished:

$\exists j, S_j^{\mu} = rsfree_0$: $uload$ is emitted for a frame $F_j$ in state $rsfree_0$. $F_j$ switches to $free_1$. When $F_j$ is enabled by $e_j$, $F_j$ executes $uload$ and switches to $active_0$. $uload$ loads an active thread $\tau$ from the memory ($T_{r_j}^{OSM}$ switches to state on processor).

$\exists j, S_j^{\mu} = free_0$: $uload$ is emitted for a frame $F_j$ in state free. $F_j$ switches to $free_1$. $F_j$ executes the routine freeTag later on and switches to $rsfree_1$. This situation appeared already above.

$\exists j, S_j^{\mu} = stopped_0$: $uload$ is emitted for a frame $F_j$ in state stopped. $F_j$ switches to $stopped_1$. When $F_j$ is enabled by $e_j$, the routine writeOut is executed by $F_j$ and $F_j$ switches to $rsfree_1$. This situation appeared already above.

3 Proving the Correctness of the Models

3.1 Proving the VTM

While in the VTM the threads are never interrupted, the corresponding thread in the OSM/MPM can be interrupted and moved to the main memory for some time. Hence, we cannot compare
the sequences of the computations step by step; instead all changes in the sequences must be consistent, i.e. we have to either remove parts in the sequences in the OSM/MPM where the instruction is not executed or we have to replicate parts of the sequences in the VTM.

For this reason, we introduce an intermediate model $\Delta VTM$ to avoid this problem. $\Delta VTM$ is in principle the same as the VTM, except that only one of the threads is able to execute an instruction at each point of time (interleaved execution). This is achieved by introducing an ‘enable’ signal for each thread frame in the VTM that allows to execute an instruction. We do not consider special interleaving techniques [1, 6], rather we only demand that at most one of the enable signals holds at each point of time and that each enable signal holds infinitely often. Now the following verification problems are to be solved:

1. We must show that $\Delta VTM$ is correct, i.e. OSM/MPM implements $\Delta VTM$.

2. The introduction of the enable signals has not affected the behavior of the VTM, i.e. $\Delta VTM$ implements VTM.

When there is an upper bound for the number of threads in the OSM, the model of the last section reduces the above correctness problems to finite state problems. These can then be solved by symbolic traversal procedures as implemented e.g. in the SMV model checker [4].

As already indicated, $\Delta VTM$ is implemented in terms of the OSM/MPM. Either the thread is not loaded on the processor, then the state of the OSM is exactly the state of the $\Delta VTM$. Otherwise, some subset of the states of the MPM define the state of the thread frame in the $\Delta VTM$. The formal definition along with the definition of the signals at the level of $\Delta VTM$ is given below:

$$
S^\Delta VTM_i := \begin{cases} 
\text{free} : S^O_i = \text{free} \lor \bigvee_{j=1}^{m} \tau_j = i \land S^\mu_j \in \{ \text{free}_0, \text{free}_1, \text{free}_2, \text{free}_3, \text{free}_4, \text{free}_5 \} \\
\text{stopped} : S^O_i = \text{stopped} \lor \bigvee_{j=1}^{m} \tau_j = i \land S^\mu_j \in \{ \text{stopped}_0, \text{stopped}_1, \text{stopped}_2, \text{stopped}_3, \text{stopped}_4, \text{stopped}_5 \} \\
\text{active} : S^O_i = \text{active} \lor \bigvee_{j=1}^{m} \tau_j = i \land S^\mu_j \in \{ \text{active}_0, \text{active}_1, \text{active}_2, \text{active}_3, \text{active}_4, \text{active}_5 \} 
\end{cases}
$$

$$
\text{getfr}^\Delta VTM_i := \bigvee_{j=1}^{m} \tau_j = i \land S^\mu_j \in \{ \text{free}_0, \text{free}_1, \text{free}_2, \text{free}_3, \text{free}_4, \text{free}_5 \} \land \text{getfr}_j
$$

$$
\text{reldr}^\Delta VTM_i := \bigvee_{j=1}^{m} \tau_j = i \land S^\mu_j \in \{ \text{stopped}_0, \text{stopped}_1, \text{stopped}_2, \text{stopped}_3, \text{stopped}_4, \text{stopped}_5 \} \land \text{reldr}_j \lor \tau_j = i \land e_j \land I^\mu_j = \text{reldr}(i)
$$

$$
\text{start}^\Delta VTM_i := \bigvee_{j=1}^{m} \tau_j = i \land S^\mu_j \in \{ \text{stopped}_0, \text{stopped}_1, \text{stopped}_2, \text{stopped}_3, \text{stopped}_4, \text{stopped}_5 \} \land \text{start}_j
$$

$$
\text{stop}^\Delta VTM_i := \bigvee_{j=1}^{m} \tau_j = i \land S^\mu_j = \text{active}_0 \land e_j \land I^\mu_j = \text{stop}
$$

The definitions above are well-defined and implement VTM, as the following properties can be proved (this is part of our specification):

- The cases in the definition of $S^\Delta VTM_i$ are distinct and cover all situations, i.e. all reachable states of the OSM and MPM.
- At most one of the signals getfr$^\Delta VTM_i$, reldr$^\Delta VTM_i$, start$^\Delta VTM_i$, and stop$^\Delta VTM_i$ does occur at a point of time.
The signals \( \text{getf}_i^{\Delta VTM}, \text{relf}_i^{\Delta VTM}, \text{start}_i^{\Delta VTM}, \text{stop}_i^{\Delta VTM} \) are only generated, when the state \( S_i^{\Delta VTM} \) is able to react on them according to figure 1.

- The state transitions as given in figure 1 do hold and there are no more transitions.

### 3.2 Proving Fairness

The verification of the specifications in the last subsection assures that \( \Delta VTM \) is an abstraction of OSM/MPM. However, this does not assure that anything happens at all. Instead it only assures that if something happens, it will behave as expected. In particular, we do not know, if all threads are executed. This essential property has be verified additionally.

However, the fairness of OSM/MPM depends crucially on the routine \( u\_\text{load} \), i.e. we have to assume that \( u\_\text{load} \) selects the thread frames to be loaded on the processor in a fair manner. This means if \( u\_\text{load} \) would be called infinitely often, then it must choose each thread frame that is in state \( S_i^{OSM} = \text{active} \). Formally, this assumption \( u\_\text{load} \_\text{fair} \) is defined in CTL [3] as given below.

\[
\text{u\_load\_fair} := (AGAFu\_load) \rightarrow \bigwedge_{i=1}^{n} AG \left( S_i^{OSM} = \text{active} \rightarrow AF \left( S_i^{OSM} = \text{on\_processor} \right) \right)
\]

\[
\text{fairness} := u\_\text{load}\_\text{fair} \rightarrow \bigwedge_{i=1}^{n} AG \left( S_i^{OSM} = \text{active} \rightarrow AF \left( S_i^{OSM} = \text{on\_processor} \right) \right)
\]

Under this assumption \( u\_\text{load}\_\text{fair} \), we can prove that no thread that has been written into the memory in state \( S_i^{OSM} = \text{active} \) will be left out there forever. If \( u\_\text{load} \) is called infinitely often (i.e. \( AGAFu\_\text{load} \) holds), then it can be obviously seen that the specification \( \text{fairness} \) can be derived from our assumption \( u\_\text{load}\_\text{fair} \). However, it remains to show, that either \( AGAFu\_\text{load} \) holds or if there is a point of time \( t_0 \) such that \( u\_\text{load} \) is never called after \( t_0 \), then all thread frames in the OSM will never be in state \( \text{active} \) after \( t_0 \).

### 3.3 Experimental Results

If we assume an upper bound for the number of existing threads at each point of time, our model yields in a reduction of the verification problem to finite state problem. We have proved the correctness and fairness of the implementation of Rhamma’s thread manager by the SMV system [3]. Table 3.3 gives the resource requirements for \( n \) thread and \( m \) register frames we obtained on a UltraSparc with 196MByte main memory under Solaris 2.5.1.

Although we cannot claim to have used optimal variable orderings that are essential for BDD based methods, it seems that the problem is hard for symbolic traversal techniques. For this reason, we have verified the model using the interactive theorem prover HOL[5] for arbitrary numbers of threads and register frames. The correctness of the VTM was thereby straightforward and required only simple term rewritings.

### 4 Conclusions

We have shown how thread managers of multithreaded systems can be modeled in an abstract way and verified separately from the rest of the system. While the description of the thread management retains most of the details of the system, only a few assumptions on the environment of the thread manager are necessary. These assumptions concern essentially the implementation of the interrupt service routines. In particular, in case of Rhamma, that has been used as a case study throughout this paper, we could reduce the fairness of the entire system to the fairness of the interrupt routine \( u\_\text{load} \).
The derived assumptions on the interrupt service routines can directly be used in a modular verification as specifications for these routines, when the entire system is to be verified. The abstraction to the thread manager is mainly done by abstracting from the programs that actually run in the threads and by abstracting from the interrupt service routines.

The model can be used for formal verification. It allows, to reduce the verification problem to a finite-state problem under the assumption that not infinitely many threads are generated. Our method allows hence the use of common model checkers to prove the thread manager. In case of Rhamma, however, this was only for small numbers of threads possible. Hence, we used the HOL theorem prover to assure the correctness, and the model proved to be useful also in that context.

Nevertheless the automated verification using SMV was helpful in defining a correct model for the processor and also influenced the design in an early phase. It was surprising that even after lots of simulation runs, SMV has found a subtle error in the timeout mechanism. Once detected, it was easy to correct the error.

References