Abstract—Future embedded system products, e.g. smart hand-held mobile terminals, will accommodate a large number of applications that will partly run sequentially and independently, partly concurrently and interacting on massively parallel computing platforms. Already for systems of moderate complexity, the design space will be huge and its exploration requires that the system architect is able to quickly evaluate the performances of candidate architectures and application mappings. The mainstream evaluation technique today is the system-level performance simulation of the applications and platforms using abstracted workload and processing capacity models, respectively. These virtual system models allow fast simulation of large systems at an early phase of development with reasonable modeling effort and time. The accuracy of the performance results is dependent on how closely the models used reflect the actual system. This paper presents a compiler based technique for automatic generation of workload models for performance simulation, while exploiting an overall approach and platform performance capacity models developed previously. The resulting workload models are experimented using x264 video and JPEG encoding application examples.

I. INTRODUCTION

The functionalities of currently independent mobile phones, music and movie players, portable televisions, and internet tablets are converging to one multi-function device. Future multi-processor designs will be massively parallel systems consisting of several heterogeneous subsystems containing processors, memories, and peripheral interfaces. The Network-on-Chip communication paradigm will be used between the subsystems for increased scalability, even though it will bring uncertainty with respect to latencies in case large centralized memories are needed.

System complexity will increase by orders of magnitude within a decade as a result of these developments. The embedded system products are usually constraint with strict real-time, energy and resource budgets. Moreover, application and platform designers will have a considerable number of design alternatives. In the future, novel systematic approaches will be needed for design space exploration. Efficient methods and tools will be required to avoid wrong design decisions in the critical stages of development.

Performance evaluation has been approached in many ways at different levels of refinement. SPADE [1] implements a trace-driven, system-level co-simulation of application and architecture. Artemis [2] extends this by introducing the concept of virtual processors and bounded buffers. TAPES performance evaluation approach [3] abstracts the functionalities by processing latencies and covers only the interaction of the associated sub-functions on the architecture without actually running the corresponding program code. MESH [4] looks at resources, software, and schedulers / protocols as three abstraction levels that are modeled by software threads on the evaluation host. SysXplorer [5] abstracts communicating processes in SystemC and analyses communication for non-pre-emptive software scheduling.

Workload models are an important part of the performance evaluation and should provide a proper abstraction of the actual applications. The term workload usually refers to computational workload and is almost synonymous with the set of traditional computer benchmarks that have been used effectively to measure and compare the computational effectiveness of various computer architectures in architectural simulations [6]. Targeted more to system-level [7] describes a multi-programmed workload model to estimate the performance and energy consumption of a CMP architecture, [8] presents a method to characterize properties of multimedia workload relevant to SoC platform design, and [9] proposes synthetic methods based on either application workloads or real applications to model concurrent execution of applications for the Sesame framework.

The performance modelling and evaluation approach of the authors [10] differs from the above approaches as to the way the application is modeled and abstracted. The workload model mimics truly the control structures of the applications, but the leaf level load data is presented like traces. Also the execution platform is modeled rather at transaction than instruction level. The timing information is resolved in the simulation of the system model in our case, which differs from the technique used in [11]. Several methods exist for obtaining the load information for workload models. The analytical method is described in [12] and trace- or measurement-based approach in [13].

The contribution presented in this paper is an automatic workload model generation for system-level exploration based on a modified compiler. It is described in the context of the overall approach and platform performance capacity models developed previously [14]. The compiler-based approach generates models for performance simulation from source code. The approach is experimented with using an x264 video and JPEG encoding application examples.

The rest of the paper is structured as follows: Chapter II out-
lines the models used in the exploration, Chapter III describes the compiler based workload model generation, Chapter IV presents case studies, and Chapter V draws conclusions.

II. MODELS IN SYSTEM-LEVEL EXPLORATION

The ABSOLUT performance simulation approach [10] abstracts the application models as workload models and the execution platform models as capacity models. The workload models are allocated to the platform models and simulated at the transaction level to obtain performance data.

The purpose of workload modeling is to illustrate the load an application causes to an execution platform when it is executed. Workload models do not perform the calculations or operations of the original application. However they enable performance evaluation already in the early phases of design because applications need not be finalized. Workload modeling also enhances simulation speed as the functionality is not simulated and models can typically be easily modified to quickly evaluate various use cases.

The workload models have a hierarchical structure [10] (figure 1), where top-level workload model divides into application workload models and common control, which takes care of the concurrent execution of the workloads. Similarly, each application workload is constructed of one or more processes and control. The processes, in turn, are comprised of function workload models and control. The control is implemented using standard C++ control structures in SystemC based workload models. Function workload models are basically control flow graphs with basic blocks and branches; basic blocks are ordered sets of workload primitives used for load characterization. Workload primitives are abstract instructions read and write for modelling memory accesses, and execute for modelling data processing.

Platform modeling comprises the description of both hardware and platform software (middleware) components and interconnections that are needed for performance simulation. The platform model contains cycle-approximate timing information along with structural and behavioral aspects. The platform model is composed of three layers [10]: component layer, subsystem layer, and platform architecture layer. Component layer consists of processing (e.g. processors, DSPs, dedicated hardware and reconfigurable logic), storage, and interconnection (e.g. network structure and bus) elements. The subsystem layer is built on top of the component layer and instantiates a set of components and their connections. The platform architecture layer incorporates the subsystems with platform software and serves as the portals that link the workload models and the platforms in the mapping process. Each layer has its own services, which are abstraction views of the architecture models. They describe the platform behavior and related attributes, e.g. performance, but hide other details. The platform model can be constructed from a existing library of performance models of components. The resulting models provide interfaces, through which the workload models use the resources and services provided by the platform.

In the allocation phase, each workload model entity is linked to a processor or other component, which is able to provide the services required by that entity. After mapping the workloads to the platform, the models can be combined for transaction-level performance simulation in SystemC.

The output of simulation consists of the total execution time for the simulated use case, and output from the performance probes included in the workload and platform models. Typically the probes are used to extract:

- The utilization of components as a function of time, and averaged across the entire simulation run. E.g. the time a processor spent in busy state or waiting for I/O.
- I/O traffic initiated, transferred, or serviced by the components. E.g. how many read and write requests were handled by a specific memory block.
- The processing time of services (average, minimum and maximum). E.g. how many milliseconds did it take for a video accelerator to encode one frame of video data.

Based on the utilization of a processor model in the platform, the designer can for example evaluate, whether a faster processor or a slower (lower-power) processor would be necessary, or if clock frequency and / or voltage changes would be enough to meet the requirements. Based on the application execution time and performance data collected from the platform, it is possible to consider if it is the execution capacity of the platform that should be improved, or if the application should be utilizing the platform more efficiently. If necessary, the system level exploration is iterated back to application or workload or platform modeling.

III. ABSINTH

ABSINTH (ABstract INstruction exTraction Helper) is a tool for generating workload models from application source code. The generated models are intended for performance simulation with the ABSOLUT approach (figure 2).

ABSINTH has been implemented by extending GNU Compiler Collection (GCC) version 4.3.1 with two additional passes (figure 3). It can be triggered with a single switch (-fabsinth) during the compilation of any source code supported by GCC. The first ABSINTH pass, pass_absinth_cf, is responsible for constructing the function layer of the workload model, i.e. the control flow between basic blocks in each source code function. The second pass,
pass_absinth_bbs, will traverse RTL (GCC’s low-level intermediate language) to extract load primitives read, write, and execute for each basic block. The load primitives correspond to memory loads, memory stores, and data processing or control instructions respectively. Both passes are located near the end of the optimizations, between pass_rtl_loop_init and pass_rtl_move_loop_invariants [15] in the present version of the patch. The rationale for the location is the following:

1) Passes must be near the end to easily detect memory reads, writes and data processing instructions. This also ensures that the models resemble the real, compiled application as much as possible.
2) The information on loops (which blocks belong to which particular loop) must still be present.
3) Passes must be near each other so that the control flow does not change in between.

ABSINTH generates workload models at a late phase of compilation after most optimization passes. Thus the resulting models are somewhat target dependent. For best accuracy one should use the same compiler target architecture in model generation and performance simulation (e.g. a cross-compiler).

There are three phases in the model generation process to obtain a proper model of control for the models (figure 2):

1) First, the source code must be compiled with profiling (-fprofile-generate).
2) then, the compiled binary must be executed with a data set corresponding to the use case, and
3) finally, the source code must be compiled again with both profile-guided optimization and ABSINTH enabled (-fprofile-use -fabsinth)

ABSINTH uses the profiling data during model generation for probabilities of branches, which are modeled statistically. It is also used to extract the number of iterations for loops.

ABSINTH does not have dependencies on any particular source code language, so it should be able to generate workload models from any language supported by GCC. However, only the C front-end has been used so far.

A. Control flow

pass_absinth_cf generates the function layer of the workload model by traversing the loops, basic blocks, and basic block edges in the GCC’s control flow graph of the source code. For each basic block (BB), it will generate a call to the workload model of the basic block, if the BB does not belong to any loop (figure 4). It will also process the block exit edges by generating a probability-based test and conditional jumps to the target blocks. During simulation the test will be executed and one of the edges selected accordingly. The probabilities of the edges are obtained from profiling data. On the other hand, if the BB is a loop header (entry) block, pass_absinth_cf will generate a call to the control flow model of the loop and then handle the loop exit edges like above. Otherwise, it will just ignore the basic block and continue with the next one.

The probability-based approach is used to model if- or switch-like branches within the control flow of the application. The same approach could also be used for modeling loops: the exit condition of the loop would be just one branch with exit edges to the beginning of the loop and out of it. Unfortunately, due to the use of probabilities, it would result to a very unreliable model. The number of iterations would vary from one simulation to another and only occasionally would it hit the real value. Thus, loops are handled separately.

pass_absinth_cf needs to obtain the number of iterations for all loops. GCC already knows it for loops, where the number is constant, but for data-dependent loops it must be calculated from profiling information. Let us denote the number of times a loop header block has been executed with \( h \). \( b \) corresponds to the number of times any loop block not belonging to subloops (inner loops) has been executed. Such a node is executed only once per each loop iteration. Thus, on average, the loop has

\[
I = \frac{b}{h}
\]  

iterations. pass_absinth_cf generates a for loop, which repeats the control flow within the loop \( I \) times.

For each loop, pass_absinth_cf will generate a model of the control flow with a method, which is similar to that of figure 4. First, it will generate a call to the workload model of the
loop header block. Then, it will go through all the blocks in the loop, except header and latch (exit):

1) A function call to the workload model of the BB is generated, if the node does not belong to subloops
2) A function call to the control flow of a subloop is generated, if the node is the header block of the subloop
3) Otherwise, the BB is ignored.

Exit edges from BBs or subloops are handled as in figure 4. Finally, a call to the model of the loop latch block is generated.

B. Workload primitives

pass_absinth_bbs traverses all RTL expressions [15] within each basic block. For each expression, pass_absinth_bbs analyses, whether it is an instruction or not, and generates one execute primitive per each instruction (table I). Furthermore, it evaluates, whether the instruction will result to memory being read or written, and generates a read or write primitive respectively. From CALL_INSN expressions the name of the called function is obtained, and a call to a workload model with the same name is inserted in the generated model. At the moment ABSINTH is not able to obtain function names for calls performed via function pointers. Thus, a placeholder for the function call is generated and the correct function or functions must be inserted manually.

Finally, pass_absinth_bbs merges consecutive read, write, or execute primitives into one — i.e execute(); execute(); execute() becomes execute(3) — to reduce the size of the workload model and speed up the simulation. Alternatively, if -fabsinth-bbs-simple flag is used during the second compilation phase, the entire basic block is coalesced into one read, one execute, and one write primitive in that order. The size of the model decreases but at the cost of accuracy.

The following example contains the workload primitives of one basic block inside the DCT algorithm of a JPEG encoder. m_host is a pointer to the platform model’s component, which provides the primitive read, write, and execute services to the workload. The interfaces between ABSOLUT workload and platform models have been presented in [10].

```c
inline void T1_dct_WL::node_3()
{
    m_host->execute(1);
    m_host->read(m_read_addr, 1, 8);
    m_host->execute(2);
    m_host->write(m_write_addr, 2, 8);
    m_host->execute(2);
}
```

C. Postprocessing of generated models

ABSINTH generates one workload model per each function in the application source code. All function workload models can contain several calls to other function workloads, but they do not know where the implementations of those workloads are. ABSINTH manager is a Python script, which is able to detect the function dependences in a set of function workloads and modify the files in a way that they are properly linked and can be compiled for simulation. ABSINTH manager also warns, if the implementation of any of the function workloads is missing, and it can optionally generate a model stub for each missing function. The missing functions result from the fact that typical source code contains library function calls, which will not be compiled at the same time as the application. Thus ABSINTH will not generate models for those functions. The models can be created by compiling also the library with ABSINTH, or by extending the stubs manually. In addition, it is possible to simulate the application with the empty stubs, but it will decrease the accuracy of simulation results.

ABSINTH manager is also able to create models for processes and subthreads derived from the base class of ABSOLUT process workload model, which contains a SystemC thread where the actual processing is done. When ABSINTH-generated function workloads are used it is enough that the process workload model calls the function workload of the main() function. The models of the subthreads invoke the first function workload of the thread but are otherwise similar to the process workloads.

<table>
<thead>
<tr>
<th>RTL expression</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>JUMP_INSN</td>
<td>Generate execute primitive.</td>
</tr>
<tr>
<td>CALL_INSN</td>
<td>Generate execute primitive. Insert a call to another workload model, whose name is obtained from SYMBOL_REF.</td>
</tr>
<tr>
<td>INSN</td>
<td>Generate execute primitive.</td>
</tr>
<tr>
<td>MEM</td>
<td>Generate read or write primitive depending on whether the particular MEM expression is the first or second operand of a SET expression.</td>
</tr>
<tr>
<td>SYMBOL_REF</td>
<td>If found as a part of a CALL_INSN expression, obtain the name of the function.</td>
</tr>
</tbody>
</table>

Table I

RTL expression and corresponding action performed by pass_absinth_bbs.
IV. CASE EXAMPLES

The following case examples illustrate how ABSINTH was used to create workload models of the x264 video encoder and several parallel or sequential versions of a JPEG encoder. The execution platform model for the performance simulation of the case examples is depicted in figure 5. It consists of 4 ARM nodes connected by routers, which form a ring-like network. Each node has an ARM9 CPU, some local SRAM memory, a shared bus, and an interface to the other nodes. The accuracy of the ABSOLUT simulation approach has been evaluated with several case examples in [13], [16], [17].

A. x264 video encoder

The source code of x264 was compiled with profiling enabled and then the encoder was used to transform uncompressed video to H.264 format. In this case, a sample video file in y4m format was encoded with a target bit rate of 1000 kbit/s. The final step with ABSINTH was to recompile the x264 with both profile-guided optimisation and ABSINTH model generation enabled.

ABSINTH manager was used to link the generated function workload models so that they can properly call the other function workloads during simulation. It was also used to generate model stubs for missing functions, which in the x264 case were standard C library functions. Finally, the ABSINTH manager was used to produce a process workload model, which was subsequently mapped to the ARM9 of node 0 of the execution platform and used the local SRAM memory for storage.

Two sets of workloads were used for simulations: first normal ABSINTH models and then ABSINTH models with simple basic blocks. Five simulation runs were performed with both sets. The simulation speeds were 1/70th and 1/17th of real time respectively. The video encoding takes about 53 seconds to complete (table II). Since the ARM CPU in the first node is running the encoder as fast as it can, its utilisation is 100%. 43% of the load comes from data processing and the remaining 57% from waiting for memory read/write to complete. SRAM memory and bus load is about 32% and 31% respectively.

The simpler workload models were trading off simulation accuracy for simulation performance as expected: The simulated execution time was about 10 seconds less with the simple models mostly due to smaller SRAM utilisation, which in turn was caused by the coalesced load primitives. Despite the statistics-based approach for modelling the control flow branches, the utilisation of the components was the same across all five simulations with both sets of models.

B. JPEG Encoder

The second case example consisted of a JPEG encoding application. ABSINTH was used to generate four sets of workload models from the encoder. The first one was from the unmodified sequential application and the other three from parallelised versions of the application: Par-1 had two threads with the second thread executing Getblock and DCT algorithms. Par-2 consisted of three threads with the second and third one interleaving the execution of Getblock, DCT, and Quantization. Par-3 had also three threads with the second and third thread executing just Getblock and DCT in an interleaved manner. The models were mapped to the ARM nodes of the platform according to table III.

The mapping of the threads can be modified with a single line of code in ABSOLUT. Furthermore, altering the partitioning of the algorithms can be done quickly by modifying the parallelisation specification and then running both MPA and ABSINTH to generate another set of models.

The parallel versions of the JPEG encoder were created with the IMEC’s MPSoC Parallelization Assist (MPA). MPA is a tool for efficiently mapping applications onto multicore platforms. It takes sequential C source code and a parspec file describing its computational partitioning as input. The tool performs the partitioning, inserting communication and synchronisation as required to respect the dependencies in the sequential application [18]. The platform model contained a model of the MPA runtime library for thread management.

The execution time of the sequential encoder was about 70 ms (table IV). The Par-1 and Par-3 versions improved
it to about 55 ms, i.e. the partitioning in Par-3 (table III) did not bring real benefits compared to Par-1. However, Par-2 provided a speedup of 1.8 with an execution time of 39 ms. All speedups given by the ABSOLUT approach are lower than the theoretical maximum given by MPA high-level simulation, which assumes linear scaling and does not take the communication overhead into account.

Both Par-1 and Par-3 have 100% utilisation on the cpu of the ARM node 0. Par-1 has 44% cpu utilisation in the second ARM node, whereas Par-3 has 21% utilisation across nodes 1 and 2. Par-2 has 88% utilisation in the first node: it is idling at some point of simulation while waiting data from the other two threads. Since Par-2 has a shorter execution time and more work for nodes 1 and 2, the cpu utilisation in those nodes is considerably higher at 53%.

V. CONCLUSIONS

A compiler-based workload model generator was described for automatically creating workload models of applications for performance evaluation purposes. The generation process involves compiling application source code first with profiling enabled, then running the compiled binary with a suitable data set, and finally compiling the source code again with profile-guided optimization and model generation. As a result, one workload model, characterizing the control flow and data processing load, is created for each function in the source code. The models can be mapped to a SystemC-based execution platform model and simulated. The simulation results reveal application execution time and utilization of the platforms components among others.

The model generator was used to create two sets of models of the x264 video encoding application: one with accurate and another with simplified basic blocks. Subsequently it was used to create models of four different versions of a JPEG encoder. The models were successfully generated, then mapped to an existing execution platform model and simulated.

So far, ease of generation and debugging have had a higher priority than simulation-time performance. Thus, the automatically generated workload models are more complex in structure and slower to simulate than hand-written models. Obviously, improving this is one of the focus areas for the future development of ABSINTH.

ACKNOWLEDGMENT

This work is supported by the European Community under the grant agreement 215244 MOSART. We would like to express our gratitude to IMEC for providing us the MPA toolkit for project use.

REFERENCES


<table>
<thead>
<tr>
<th>Seq</th>
<th>Par-1</th>
<th>Par-2</th>
<th>Par-3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Execution time [ms]</td>
<td>69.5</td>
<td>55.3</td>
<td>28.7</td>
</tr>
<tr>
<td>Speedup</td>
<td>1.00</td>
<td>1.26</td>
<td>1.80</td>
</tr>
<tr>
<td>MPA HLS speedup</td>
<td>1.00</td>
<td>1.66</td>
<td>2.50</td>
</tr>
</tbody>
</table>

| Node0 utilisation | ARM 100% | 99% | 88% | 100% |
| SDRAM 59% | 55% | 43% | 53% |
| Bus 34% | 36% | 29% | 35% |

| Node1 utilisation | ARM 0% | 44% | 53% | 21% |
| SDRAM 0% | 23% | 31% | 11% |
| Bus 0% | 12% | 18% | 6% |

| Node2 utilisation | ARM 0% | 0% | 53% | 21% |
| SDRAM 0% | 0% | 31% | 11% |
| Bus 0% | 0% | 18% | 6% |

The model generator was used to create two sets of models of the x264 video encoding application: one with accurate and another with simplified basic blocks. Subsequently it was used to create models of four different versions of a JPEG encoder. The models were successfully generated, then mapped to an existing execution platform model and simulated.

So far, ease of generation and debugging have had a higher priority than simulation-time performance. Thus, the automatically generated workload models are more complex in structure and slower to simulate than hand-written models. Obviously, improving this is one of the focus areas for the future development of ABSINTH.