IST-510255
EmBounded
Automatic Prediction of Resource Bounds for Embedded Systems

Specific Targeted Research Project (STReP)
FET Open

D25 (WP7c): Analyses integrated into compiler

Due date of deliverable: July 2007
Actual submission date: January 2008

Start date of project: 1st March 2005 Duration: 48 months

Lead contractor: Heriot-Watt University Revision: 1.16

Purpose: The purpose of this deliverable is to provide a set of resource analysis features integrated into the machine code compiler described in Deliverable D24 (WP7c). We call these analyses *instrumentation* to prevent confusion with the separate resource analysis tools. They are intended to support embedded systems programmers in performance-debugging their applications and to verify the resource usage predictions generated by the full analysis suite of tools — Deliverables D05 (WP4a) [JLH07b], D11 (WP4b) [JLH07a], and D16 (WP3c) [JH07].

Results: The main result of this deliverable is a set of memory-usage and timing measurements incorporated into the machine code compiler. These have been implemented on all the chosen architectures and have been used to verify the output from the resource analysis.

Conclusion: The main conclusions are that the analyses integrated into the machine code compiler both easy to implement and use and have been very useful in the development of both the compiler itself and the resource analysis.

Project co-funded by the European Commission within the 6^{th} Framework Programme (2002-06)			
Dissemination Level			
PU	Public		⁕
PP	Restricted to other programme participants	(including the Commission Services)	
RE	Restricted to a group specified by the consortium	(including the Commission Services)	
CO	Confidential only for members of the consortium	(including the Commission Services)	

Analyses integrated into compiler

Robert F. Pointon <rpointon@macs.hw.ac.uk>
School of Mathematical and Computer Sciences, Heriot-Watt University, Edinburgh, Scotland

Hans-Wolfgang Loidl <hwloidl@tcs.ifi.lmu.de>
Institut für Informatik, Theoretische Informatik, Ludwig-Maximilians Universität, München

Norman Scaife <Norman.Scaife@lasmea.univ-bpclermont.fr>
Laboratoire LASMEA, Blaise Pascal University

Abstract

This deliverable describes how the resource-usage (instrumentation) measurements have been integrated into the machine code compiler. The project objectives are stated and brief explanations of how these objectives were met are presented. The positioning of this deliverable is discussed in Section 4. The evidence supplied in support of these statements is listed. Finally, we cite academic papers related to this work.

Major Revisions		
Revision	Date	Changes
1.15	14 Aug. 2009	table-of-contents (addressing Review Report Year 4)
1.13	27 May 2008	appendices A B on instrumentation (addressing Review Report Year 3)
1.11	6 Feb. 2008	initial version

Contents

1 Introduction

This Deliverable, D25, describes the additional resource-usage measurement analysis (instrumentation) built into the machine code compiler described in Deliverable D24 [PS08]. This should not be confused with the full resource analysis, which attempts to predict resource usage from static analysis of Hume programs, using the formal semantics as the basis for the prediction:

Resource Analysis The resource analysis for Hume code operates on the Core-HUME intermediate language. This is a desugared version of Hume code with expressions in let-normal-form (see Deliverable D11 [JLH07a]). In order to perform resource analysis, the compiler takes the type-annotated abstract syntax tree and transforms it into Core-HUME code. By distinguishing user-level variables from compiler-generated ones, and user-level let constructs from those added by the let-normal-form transformation, we ensure that the transformation is cost-preserving. The resulting code is passed through the resource-analysis proper, which has the structure of a type-inference process over an enriched type-system (see Deliverables D11 [JLH07a] and D13 [Jos07]). Annotations in these enriched types capture information about resource consumption. The same inference engine, parameterised with different cost metrics, is used for analysis of heap consumption, stack consumption and worst-case execution time. This analysis is also available as a stand-alone tool to facilitate independent development of compiler and analysis.

Instrumentation Here, we are required to add additional statements to the generated code to gather information at runtime about the amount of stack and heap used by the program in as fine-grained a manner as possible without substantially altering the resource usage. We are also required to time the operation of the program, at the very least on a box-by-box basis as well as delays, timeouts and other real-time aspects of the program's execution.

The granularity of the timings is determined by the system time-measurement granularity. This is usually about one microsecond for x386 architectures under Linux and one processor clock cycle if real-time clocks or timers are used, for example on the M32C.

2 Objectives

This work is aimed towards achieving part of the EmBounded project overall objective 6: *development of underpinning specification, implementation and support environment for the Hume language.*

For the analysis discussed here there are two overall objectives:

- Enable verification of the main resource analysis tool. We are thus required to provide as accurate measurements as possible *which are consistent with the main resource analysis operation.* Currently, there are some differences between the subset of Hume implemented by the resource analysis and by the machine-code compiler, as well as differences in implementation. These will be ironed out at a future date.
- Support embedded systems developers by providing analysis in the form of profiling information. The measurements used to support the resource analysis can also be used by applications developers to improve the performance of their applications, either by optimising memory usage or processing times.

More detailed objectives are:

1. Inject code into the machine code compiler's output which monitors stack and heap allocation.

2. Add checkpoints to the execution of the generated code allowing box execution and timeouts to be measured.

3. Provide a human-readable summary of the gathered information upon program termination.

3 Meeting the objectives

1. Measuring stack and heap usage is relatively simple since there is a central set of routines for allocating this type of memory. We can also easily classify allocations according to their intended use, for example box versus wire memory. This has been fully integrated into the compiler.

2. Care has to be taken with measuring times, since if we are too fine-grained, for example checkpointing each HAM instruction, we will skew the execution times. Currently, we target the box entry and exit points and the wire "consume" and "set pending" routines. We measure the maximum time between box/wire invocations (equivalent to the maximum `timeout` value), the maximum execution time of a box/wire (equivalent to the maximum `within` value) and count the number of box/wire invocations to allow computation of average times.

3. If instrumentation is enabled (it can be compiled out of the output altogether for efficiency reasons) then a summary is printed at the end of execution. It is sometimes necessary to force termination of the program to allow the resource usage statistics to be printed.

Appendix A shows an example of the information generated by our instrumentation.

4 Relation with other deliverables

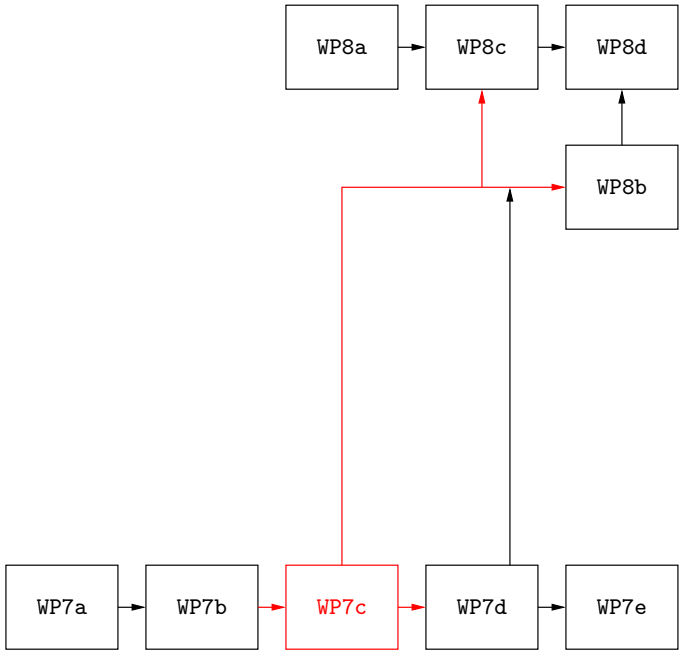

Figure 1: Inter-task dependencies for EmBounded

Figure 1 positions workpackage WP7c in the overall EmBounded task diagram. WP7c depends on WP7b (Formal translation from Hume to HAM instructions) and is required by WP7d (Hume model-checking), WP8b (Real-time computer vision algorithms) and WP8c (Evaluation of Hume). Three software deliverables have been produced in WP7c: D23 (Hume to HAM translator); D24 (Machine Code Generators); and D25 (Analyses integrated into the Compiler). A CD has been produced containing this software, and the software has also been made publicly available on the project web site (`http://www.embounded.org`).

Since work on the Hume compiler is central to the entire EmBounded project, a detailed diagram of all the interdependencies from these deliverables would be infeasibly complex. Instead, we present a table of dependencies based upon the requirements of the deliverables. Table 1 shows the deliverables which are interdependent with this deliverable.

Deliverable	Title	Notes
D03 (WP7b)	Formal translation from Hume to HAM	Compiler based upon formal translation
D05 (WP4a)	Report on stack size analysis	Compatibility with resource-analysis tools
D07 (WP8a)	Real-time testbed applications	Most of the applications work is based on the machine code compiler
D11 (WP4b)	Report on heap space analysis	Compatibility with resource-analysis tools
D12 (WP7a)	Formal semantics for Hume	Basis for the compilation scheme
D16 (WP3c)	Report on WCET analysis	Compatibility with resource-analysis tools
D23 (WP7c)	Hume to HAM compiler	Pre-processing required by the machine-code compiler
D24 (WP7c)	Machine code generators	Machine code compiler widely used by all other activities
D26 (WP7d)	Translator to support model-checking	Compatibility with verification tools
D33 (WP8c)	Report evaluating Hume	The compiler is evaluated along with the other tools in the project
D34 (WP7e)	Hume problem-solving environment	Compiler an integral part of the Hume PSE

Table 1: Deliverables dependent upon/depending on D25 (WP7c)

5 Material provided

We provide the following material as evidence of our having met the objectives in this deliverable:

- CD containing the sources and installation instructions for the `humec` compiler implementation. A single CD has been produced containing all software produced by D23, D24 and D25. This software comprises a complete tool chain taking Hume source files as inputs and generating binary files for the chosen target architecture. The tools in the tool chain are:
  - Updated implementation of `phamc`, the Hume to HAM translator with support for the `humec` compiler (D23), and including integrated analyses (D25).
  - The implementation of the HAM to C compiler `ham2c` (D24).
  - Sources for the runtime system needed by the C compiler when linking the generated object code (D24).
  - An overall management program called `humec` which calls the other tools in the chain in the right order for producing binary files.
- Installation guide and User manual for the `phamc`, `ham2c` and `humec` programs.
- Copies of the academic papers produced by the EmBounded project that are related to the development and use of the Hume to HAM translator and machine code compiler [HFH+06b, HFH+06a, Poi06, MP06].

A Sample instrumentation output

The following is a sample output from an instrumented run:

```
<<system>> hp=080ac660 sp=15 slp=6 mp=15 fp=6 rp=20 schedules=3
ctlr: state=blocked(0x0002) hpMax=175064 spMax=338 withinMax=25183 timeoutMax=5068118 active=553669 cnt=29130
circuit: state=runnable(in) hpMax=11413 spMax=341 withinMax=33540 timeoutMax=84933 active=1789998 cnt=7845
clock_mux: state=runnable(in) hpMax=2 spMax=9 withinMax=12750 timeoutMax=84976 active=1593376 cnt=7481
cmd_join: state=blocked(in) hpMax=0 spMax=8 withinMax=0 timeoutMax=0 active=0 cnt=0
cmd_split: state=blocked(in) hpMax=0 spMax=6 withinMax=0 timeoutMax=0 active=0 cnt=0
...
circuit.3->ctlr.1: avail=* pend=* hpMax=47 withinMax=1886 timeoutMax=5746564 active=12383 cnt=191
cmds.2->ctlr.0: avail=* pend=* hpMax=0 withinMax=0 timeoutMax=0 active=0 cnt=0
circuit.4->to_sim.0: avail=* pend=* hpMax=430 withinMax=8092 timeoutMax=5120668 active=44025 cnt=254
from_sim.0->circuit.0: avail=* pend=* hpMax=0 withinMax=0 timeoutMax=0 active=0 cnt=0
exit_mux.0->exit.0: avail=0 pend=* hpMax=2 withinMax=0 timeoutMax=92358356 active=0 cnt=0
command.5->exit_mux.1: avail=* pend=* hpMax=2 withinMax=1714 timeoutMax=92356634 active=1714 cnt=1
...
```

The first line gives overall values for the whole program, the second block of data is the box measurements and the final block gives results for the wires.

B Instrumentation implementation

To enable the instrumentation, the C compilation phase has to include the macro definition:

```
#define INSTRUMENTATION
```

This is actually enabled by default but can be switched off by other incompatible flags (for example the `NDEBUG` flag which disables all debug features). For compilation to the M32C, the C compiler phase is a separate tool, not under control of `humec` and the relevant definitions and macros have to be carefully checked manually prior to gathering instrumentation information.

The C structures describing boxes and wires have extra fields included if the `INSTRUMENTATION` flag is present, for boxes:

```
struct box_t {
    uint16 runPattern;
    uint16 blockPattern;
    when_t time;
#ifdef INSTRUMENTATION
    uint32 hpMax;     /* Maximum heap usage */
    uint32 spMax;     /* Maximum stack usage */
    usecs withinMax;  /* Maximum box execution time */
    usecs timeoutMax; /* Maximum time between box executions */
    usecs activeTime; /* Total time that the box was running (the RHS) */
    uint16 activeCnt; /* Incremented each time the box RHS is evaluated */
#endif /*INSTRUMENTATION*/
};
```

and for wires:

```
struct wire_t {
    wireval_t available;
    wireval_t pending;
    boolean pendingFlag;
    boolean availableFlag;
    when_t pendingTime;
    when_t availableTime;
```

```
#ifdef INSTRUMENTATION
    usecs withinMax;  /* Maximum time spent processing wire */
    usecs timeoutMax; /* Maximum time between set pending and write */
    usecs activeTime; /* Total time that it has a value within the wire */
    uint16 activeCnt; /* Incremented each time a value is output */
    uint32 hpMax;
#endif /*INSTRUMENTATION*/
};
```

To gather the instrumentation information, the key C functions for boxes are `boxBegin` and `boxEnd`:

```
void boxBegin(box_t *b) {
#ifdef INSTRUMENTATION
    usecs delta;
    delta = timeDiff(&currentTime, &b->time);
    if(delta > b->timeoutMax) b->timeoutMax = delta;
#endif /*INSTRUMENTATION*/
    b->time = currentTime;
}

void boxEnd(box_t *b) {
#ifdef INSTRUMENTATION
    usecs delta;
    delta = timeDiff(&currentTime, &b->time);
    if(delta > b->withinMax) b->withinMax = delta;
    b->activeTime += delta;
    b->activeCnt++;
    if((hp - heap) > b->hpMax)
        b->hpMax = hp - heap;
#endif /*INSTRUMENTATION*/
    b->time = currentTime;
}
```

The runtime is equiped with a suitable function for measuring the wallclock time to the finest accuracy of the system clock (`gettimeofday` for Unix-based systems and a special interrupt service routine tied to one of the timers for the M32C). These functions allow computation of all the time parameters (active, within and timeout values) and of the maximum heap usage since the heap is cleared when the box exits. For stack usage, the stack allocation routine in the runtime system is modified:

```
SP inc_sp() {
    sp++;
    if(sp > spSoftLimit) {
        throwException(currentStackException
                       ? currentStackException
                       : ((sp > spLimit) ? StackOverflow : SoftStackOverflow) );
}
#ifdef INSTRUMENTATION
    if(sp > currentBox->spMax)
        currentBox->spMax = sp;
```

```
#endif /*INSTRUMENTATION*/
    return (sp - 1);
}
```

Note that this means that the runtime system has to have been compiled with the `INSTRUMENTATION` flag enabled. Normally, the runtime system is provided as a pre-compiled library.

A similar set of modifications is required for wires, noting that wires do not actually have an associated stack.

References

[HFH+06a] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hoffmann, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards formally verifiable resource bounds for real-time embedded systems. In *Proc. Workshop on Innovative Techniques for Certification of Embedded Systems*, 2006. 5

[HFH+06b] Kevin Hammond, Christian Ferdinand, Reinhold Heckmann, Roy Dyckhoff, Martin Hofman, Steffen Jost, Hans-Wolfgang Loidl, Greg Michaelson, Robert Pointon, Norman Scaife, Jocelyn Sérot, and Andy Wallace. Towards formally verifiable WCET analysis for a functional programming language. In Frank Mueller, editor, *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, number 06902 in Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2006. Internationales Begegnungs- und Forschungszentrum fuer Informatik (IBFI), Schloss Dagstuhl, Germany. 5

[JH07] S. Jost and K. Hammond. Validation of the Prototype Worst-Case Executation Time (WCET) analysis. EmBounded Project Deliverable, October 2007. Deliverable D16. (document)

[JLH07a] S. Jost, H-W. Loidl, and K. Hammond. Report on Heap-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D11. (document), 1

[JLH07b] S. Jost, H-W. Loidl, and K. Hammond. Report on Stack-space Analysis. EmBounded Project Deliverable, February 2007. Deliverable D5. (document)

[Jos07] S. Jost. Prototype Implementation of Space Analyses. EmBounded Project Deliverable, February 2007. Deliverable D13. 1

[MP06] Greg Michaelson and Robert Pointon. Recursion, iteration and hume scheduling. In *Trends in Functional Programming (TFP'06)*, Nottingham UK, April 2006. 5

[Poi06] Robert F. Pointon. Compiling vs costing - experiences with hume. In *Implementation of Functional Languages (IFL'06)*, Budapest Hungary, September 2006. 5

[PS08] R.F. Pointon and N. Scaife. Hume Machine Code Compiler. EmBounded Project Deliverable, January 2008. Deliverable D24. 1