Profile-Based Adaptive JiT Compilation in the Context of the HiPE Compiler

Erik Andersson
eran3133@student.uu.se

May 4, 2005
Abstract

This report describes of the HiPE Tool, a general purpose profiling and optimization tool for the HiPE compiler, a high performance native compiler for the Erlang programming language. Using HiPE’s ability to compile single functions during runtime the HiPE Tool, which is programmed in Erlang, tries to compile the most called functions in an execution, in an automated fashion, to improve efficiency. Besides this, it also provides functionality for compiling, timing, profiling, to investigate function dependencies and to visualize the function dependency graphs of entire applications. This report shows the creation of, and the general functionality of this tool, compares it to other tools within the same field and does various tests.
7.2 Adding files ............................................. 18
7.3 Profiling .................................................. 19
  7.3.1 Lightweight profiling .............................. 19
  7.3.2 Heavyweight profiling ......................... 19
  7.3.3 Heavyweight or Lightweight? You decide! .... 20
  7.3.4 Native code and profiling ..................... 20
7.4 Compiling ............................................... 21
7.5 Timing .................................................. 22
7.6 Performance Metrics ................................. 23
7.7 Xref-server ........................................... 24
7.8 Investigating function dependencies ............. 25
7.9 Sizes .................................................... 26
7.10 Visual call graph .................................... 27
  7.10.1 Problems with the visual call graph ....... 28
  7.10.2 Deciding on thresholds ....................... 28

8 Advanced Functionality of the HiPE Tool ........... 30
  8.1 Interactive scenario generation .................. 30
  8.2 Batched scenario generation ...................... 31
  8.3 Which advanced mode to use? .................... 33
  8.4 Selecting profiles .................................. 34

9 Changed or removed functionality ................... 35
  9.1 Scripts ............................................... 35
  9.2 eprof ............................................... 35
  9.3 Shell-scripts ........................................ 35
  9.4 Adding files ......................................... 36
  9.5 Gathering sizes ..................................... 36

10 Comparison to similar tools ......................... 36
  10.1 VTune ............................................... 37
       10.1.1 Description .................................. 37
       10.1.2 Performance and experiences ............. 37
       10.1.3 Comparison .................................. 37
1 Introduction

Erlang is a concurrent functional programming language developed by Ericsson. Its implementation has been released as open source. It is used mainly in the telecom-industry handling such important things as routers. Erlang’s default implementation is a bytecode interpreter, making compiled code portable on all platforms where Erlang has been implemented.

HiPE is a native code compiler, developed at Uppsala University, funded by Ericsson, that compiles this bytecode into native code. This speeds things up by bypassing the interpreter to fully utilize the platforms resources. HiPE is not limited to compiling entire modules to native code, it can even compile individual functions to native code.

This paper describes the creation of an application, the HiPE Tool that has been designed to help the user efficiently utilize the HiPE compiler. This is mainly done using a greedy algorithm, by recursively compiling the most active parts of the application trying to lower the total execution time.

2 Erlang

Erlang plays an integral part of this assignment, and the following section gives a small introduction to it.

2.1 Usage

The telecommunication industry has specific needs that their software and the languages for the creation of this software need to meet. This includes demands such as: high level abstraction, correctness, fault tolerance (most downtime is due to software, not hardware errors), maintainability (ease of debugging and adding new features), soft real-time interaction (we can guarantee time-limits for actions) etc. [19]

Erlang[9] is a concurrent functional programming language developed by Ericsson to address these specific needs. Erlang is often used in various embedded environments, such as switches, where hardware and computing resources can be scarce, and conserving these resources as much as possible is of utmost importance. A prime example is Erlang’s use in the AXD301 backbone switch [11].

The primary incarnation of Erlang is Erlang/OTP which not only includes the implemented language itself, but also a wide variety of provided libraries\(^1\). Erlang/OTP has been released as open-source, but there are commercial distributions available. The difference mainly lies in the amount of provided support\(^2\).

Although initially designed for the telecom-industry, the qualities that make Erlang

\(^1\)Erlang and Erlang/OTP is often used interchangeably in this report, usually the circumstances of the text describes what is being discussed.

\(^2\)Erlang is available at http://www.erlang.se
suited for this application also make it suitable for other soft real-time industrial applications. It is especially helpful for distributed applications - much thanks to ease at which applications can be spread out over a number of nodes, the portability of the code and the ease of communicating between processes (running on different nodes).

Areas where Erlang is currently being used include [12]:

- Telecommunication systems (as described above)
- Database applications
- Servers for Internet applications

2.2 Running Erlang

Erlang is run in a runtime environment - a virtual machine\(^3\), called functions are loaded into the environment and then executed.

Erlang distributes work inside the shell using lightweight processes, and the processes are run concurrently. These processes each require a minimum of memory, and the creation and deletion of these processes (and thereby the threads) is a very inexpensive operation. Actual context switching between these process are of 1-2 magnitudes cheaper than switching threads in C. [12]

A very common approach in a number of applications is starting some sort of main-process, and then spawning in the upwards of hundreds of executing processes, who are concurrently doing different tasks. The act of spawning a new process on a remote node is almost trivial, and is often done for a variety of reasons. To guarantee that the executing processes can communicate, Erlang provides a very easily used message passing system, where the user can specify the destination and the message. These messages are sent asynchronously, Erlang does not wait for a response but instead continues executing immediately after the message has been sent. To handle these communications, Erlang has a built-in sense of time, it is for instance possible to specify how long to wait for a message before acknowledging that something has gone wrong.

Erlang uses a module-system, dividing the program into smaller more manageable pieces. This also means that it is possible to incrementally build up an application, starting with any number of modules and adding them as needed. In each of these modules, there are a variety of functions, and the user can then specify which of these functions are exported, i.e which are accessible from outside the module. Calling a function in a separate module is the simple act of specifying the module, the function-name and what arguments to pass. The function is then executed (if the module can be found and we have specified the correct arguments, the shell will load the module if it isn’t already loaded) and the result returned.

\(^3\)Abbreviated VM.
2.3 BEAMing with joy

Programs in Erlang are compiled by the normal Erlang/OTP-compiler into so called BEAM-files\(^4\) These are then loaded by the Erlang/OTP-loader into the runtime environment and then interpreted and executed there. This idea is based on virtual machine bytecode in the same way Java handles virtual bytecode. Bytecode has the advantage that it will run on any platform where the Erlang/OTP runtime environment is implemented.

2.4 Programming in Erlang

When it comes to writing actual code, Erlang is quite different from other, popular languages such as C or Java. While these are imperative programming languages, Erlang is a functional programming language, that is, the programs are built by a multitude of functions, instead of relying on the execution of commands. Functional languages spend much of its time evaluating functions, even providing functions as inputs to other functions - and functional languages do not use variables whose values can change, in stark contrast to imperative languages [13].

Erlang allows the creation of variables, but they can only be given a value once, the property of single assignment [9], meaning they are in actuality identifiers, used to give a shortcut to the return-value from a function or the provided arguments to the function. This means that things such as loops are impossible, and instead recursion is used when the program in some way need to repeat a procedure or traverse a collection of data. The need for recursion means that any heavily used function is very often a recursive function, leading to this function calling itself a great number of times, making it readily apparent where much of the time is spent (i.e, the functions that are called often).

Erlang uses dynamic typing (attempts have been made to use various forms of type checking, the most successful being the Dialyzer [17]) – the user never needs to explicitly define the types (integer, bool etc) of variables; this is instead checked at runtime. Memory-wise, the user is never forced to allocate memory, this is all done by the Virtual Machine itself during runtime. The Virtual Machine also handles garbage collection without the users supervision.

2.5 Specifying a function

In Erlang, the naming of functions is somewhat different from the majority of other programming languages. The user is allowed to have several functions with the same name, as long as they take in a different number of arguments; their arity\(^5\) differs. Inside a module the user can specify what functions he wants to call by simply calling it with functionname(arguments), and what exact version of the functions with the

---

\(^4\)BEAM stands for Bogdan/Björn’s Erlang Abstract Machine.
\(^5\)Arity is a mathematical term for the number of arguments a function takes
same name is figured out by investigating the number of given arguments. This leads us to fully specify a function by specifying what module it appears in, what name it has and what arity the version has, often represented as a tuple: \{Module,Function,Arity\}.

3 HiPE

HiPE is a native code compiler for Erlang, and is the other basis for this report.

3.1 Introduction to HiPE

HiPE \[15, 22\] is short for High Performance Erlang which is a just-in-time native code compiler for the Erlang/OTP system (described in section 2). The goal of HiPE is to be able to translate BEAM bytecode, that has been generated to be run by our virtual machine, into native code for the target platform, to increase performance. HiPE currently has working backends for SPARC V8+, x86, AMD64 and PowerPC. It exists as a new component of, and is fully integrated in, later versions of Erlang/OTP.

Not detailing exactly how HiPE works, there are a few things that are important to get a working context for this report. HiPE is created in such a way that having both native code and bytecode in the same application is perfectly fine, meaning it is never guaranteed (nor is the user forced to) to have the entire program in native code. As both native and emulated code may be running at the same time it is often necessary to switch between them. Whenever such a switch happens, there is a cost involved. Even if the switches are handled completely transparently, knowing that they’re expensive means the user should take steps to avoid mode-switches whenever possible.

Which brings up another key-feature; the HiPE-compiler is very flexible as to how, what and when it can compile. We are not limited to directly compiling to native code using the actual source-files themselves, we can instead use the BEAM-files to generate the native code. The compilation can also be done in the actual runtime environment, as programs are being executed, this is a just-in-time compiler, where we can actively compile even as the system is up and running. After compiling, the new native code is loaded in and any calls to the now native code are trapped and sent to where the new code is located. As we compile modules we can either do this strictly in the current runtime, if this is restarted the code would be loaded as bytecode again, or we can save the native code to BEAM-files, creating fat files which may contain native code for several platforms. Besides the obvious flexibility of the described system, there is one feature that greatly enhances this idea; we are not forced to compile entire modules, the granularity of the system is that of single functions. This means we can specifically decide which functions of our application should be compiled to native, giving us much greater control over how we chose to compile our application.
3.2 HiPE options

As most compilers, the HiPE native compiler allows the user to use a variety of options (or flags) to specify the compiler’s behavior [22]. By changing these flags one can either have a faster compilation that possibly results in less optimized code, or we may use more advanced options to try to create better and more efficient code, but taking a longer time to compile. To help the users there are several levels of pre-defined optimization available; o1, o2 and o3, where o3 is the most advanced and o2 is the default. These are not really compiler options in themselves, but merely macros for a set of pre-selected flags.

4 Terminology

This report uses a number of terms that need to be defined.

4.1 Hotspot

A hotspot is a function where a great amount of time is spent while executing an application. These hotspots are obviously where most of our efforts should be spent to increase the efficiency of the code (in the case of the HiPE Tool, compiling them to native code).

4.2 Descendants

The descendants of any given function (called the parent) are the functions that in some way end up being called by the parent. That is, the descendants are the functions that are directly called by our parent, and the functions that in turn are called by these directly called functions and so on until a stable set of functions has been reached.

If the parent is defined as $P$, the set of all functions (excluding $P$ - as a function can not a descendant to itself) in the set of all interesting modules is defined to be $F$ and the function $Desc(x, y)$ returns true if the function $x$ calls on the function $y$, then the set of descendants of a function can be expressed mathematically as:

\[ F := \text{All functions excluding } P \quad (1) \]
\[ D_0(P) = \{ \forall x \in F : Desc(P, x) \} \quad (2) \]
\[ D_{n+1}(P) = \{ \forall x \in F, \exists y \in D_n(P) : Desc(y, x) \} \cup D_n(P) \quad (3) \]
\[ \text{End if: } D_{n+1} = D_n \quad (4) \]

This is a recursive definition, new functions are added to the list of descendants in each iteration and the function terminates when no new descendants were added in an iteration. As the parent was explicitly removed from the list of functions to consider, it is never included in the list of descendants (even if the function is recursive and calls
It is however entirely possible to have two functions who are both descendants of the other.

An example in Erlang code:

\[
a() \rightarrow b() + c().
\]

\[
b() \rightarrow d().
\]

\[
c() \rightarrow e().
\]

\[
d() \rightarrow 1.
\]

\[
e() \rightarrow 1.
\]

If a() is our parent, the functions b(), c(), d(), e() are all descendants, as they all in some way end up being called after initially calling a(). b()'s descendants on the other hand is the single function D().

4.3 Compilation scenario

A "compilation scenario"\(^6\) describes how functions are compiled for an application, it is a list of what functions are in the native compiled mode, and which are in emulated bytecode.

In a more mathematical terminology this can be expressed as the following: given a set of modules \( M \) with a number of functions in them denoted by \( F \); a compilation scenario is the partitioning of these functions \( F \) into two separate classes, \( F_B \) and \( F_N \). \( F_B \) are the functions that are in emulated bytecode, while \( F_N \) are those functions that have been compiled with the HiPE compiler and are run in native mode. A detailed compilation scenario is then a scenario where one, besides specifying which functions are in native, and which are in bytecode, also associates with each function in \( F_N \) a list of the specific compiler options used to compile this function.

A shorthand version of a detailed compilation scenario is to only specify the functions that are in native, and the functions used to compile these (see example below), \( F_B \) is not specified and this set is simply deemed to be \( F \setminus F_N \).

Example of a detailed compilation scenario (shorthand):

\[
\begin{align*}
\text{foo:bar/1,}[o3] \\
\text{foo:bar/2,}[o2] \\
\text{moo:sort/1,}[o2]
\end{align*}
\]

\(^6\)Often just specified as scenario.
4.4 Function dependency graphs

Function dependencies describe how functions interact with one another. More specifically, the dependency graph shows how functions call each other, something that is often very interesting when it comes to profiling programs. If it is possible to find out what functions are being called by what functions; this can be used to find the descendants of a specific function, which, in the context of the HiPE Tool is quite necessary.

There are really two different function dependency graphs:

1) The static dependency graph is the information that can be gathered by a simple investigation of the source-files of an application. By reading the written code, it is possible to see what functions are called in each function, making it possible to build up a graph of these function dependencies. This dependency graph then details all the functions that might end up being called in relation to those who might call them.

2) The dynamic dependency graph, on the other hand, is similar to the static dependency graph, but functions that were not actually called in the execution of the program (or those where the number of calls were below a specified threshold) are filtered out. This can be done by combining the static dependency graph with the information gathered from profiling the execution of the application in question. The reason for constructing a dynamic dependency graph is simple; those functions that were not called in the normal execution of the program are most likely functions that are very seldom used or those that handle critical errors in the code (for instance a call to exit the program if an incorrect type was retrieved). These functions are often not of interest when it comes to decreasing the execution time of the application.

There are a number of special cases in Erlang that is of interest when trying to create dependency graphs. The first one is the function apply(M,F,A) that calls the function specified via the M,F,A tuple. The way the system is designed, the function called will not be counted. Secondly, if there are functions that retrieves the module name via an argument and then tries to call a general function in it (usually module_info()), these are also not included in the graph. Third, higher order functions, for instance various list functions, that can use functions as their arguments, are counted, but the argument functions are not.

4.4.1 Visualization of function dependency graphs

As function dependency graphs are in essence just that, graphs, it is simple to use the information gathered to visually investigate our application. In the visual representation the following is done: each function is represented by some sort of geometric shape (in the case of the HiPE Tool, a rectangle) and the fact that one function is called by another is shown by a directed edge (ending in the function that was called). The user is then presented with a very simple to use graphical overview of how a part of his program works. In the most simple case a single module might be investigated, while in very large applications hundreds or thousands of functions may be involved.
4.5 Entry point

An entry point is a specific function in a specific module with specific arguments that are used to start the application the user is trying to process. In the case of the HiPE Tool for example, the user starts the program by calling: tool:start().

5 The underlying problem

The HiPE compiler is a powerful tool in itself, it often allows the user to improve the performance of his or her application without rewriting the code.

As any tool, the HiPE compiler needs to be used right, actually deciding what specific functions (or modules) should be compiled to native code is not a trivial decision. Not all code might gain something from being compiled to native, a good example is the code that calls built in Erlang/OTP-functions (as these functions are always implemented in C) or those functions that do very little computational work [22]. Compiling with the HiPE compiler also increases the size of the loaded code, which is a punishing factor against indiscriminate compilation to native code.

The functions that actually should be compiled with HiPE are the ones that are called often and do actual work (i.e not just waiting for messages). It is also important to avoid mode-switches (see section 3.1). So what is wanted is to compile the most called code and also compile all of these function’s descendants to guarantee that once we enter native code we stay there for as long as possible, avoiding the cost of switching modes. Having all of this come together manually is a frustrating operation; not only do we want to time our application, we also want to profile it to see where the most time (and usually the most calls are spent), and then compile and see if a better result was achieved.

5.1 Why can’t I just compile everything to native?

You can if you want to. But one large problem here is that the native code requires more memory than the BEAM-code, and together with the small caches most processors have we may suffer from excessive cache misses [22].

Generally, for smaller applications compiling all the modules is quite possible, but as the size and complexity of the applications grows, the more we need to compile, and the longer it will take (compiling a module can take anywhere from a few hundred milliseconds to several seconds, depending on the system and the size and complexity of the module – punishing if you want to do this during runtime and with many modules). Also, as compiling to native code incurs an increase in loaded code size this might very well be prohibitive for larger applications (especially if used in a limited memory environment). To avoid this, it is generally a better idea in most larger and complex applications to compile to native just those functions that are called most often (and their descendants to avoid mode switches), which saves us time and code size, while still producing good results.
5.2 The Basic ideas for the HiPE Tool

Using a powerful native compiler in an efficient manner is not easy to do. With the HiPE Tool, the user is presented with an overview over his modules, with interesting information and a system that tries to iteratively use the HiPE compiler to gain the greatest speedup, while compiling as few functions as possible.

This should be done by trying to run the target application, profiling it and then trying to compile those parts of the program that do most work and where we should gain the most by increasing efficiency. This is in essence a simple idea, but it is more nuanced than that, as we do not just want to compile specific functions but also their descendants to avoid mode-switches and we are also interested in outside functions that should be used.

6 Modules used in HiPE Tool

In the Erlang/OTP system there are a lot of provided modules that do specific tasks, which the HiPE Tool relies quite heavily on. Besides using modules provided in the Erlang/OTP system the HiPE Tool also uses a few other outside applications. In this section most of these modules and applications are reviewed and explained.

6.1 xref

Xref [7] is a tool, supplied in the Erlang/OTP distribution, for investigating dependencies between modules, applications and functions. It demands that the user creates a server, to which he can then add entire applications, directories or modules (with some demands on how these are compiled to give complete information) to be investigated. Using a small set of functions, it is then possible to retrieve information on how these modules, functions, directories etc, interact with each other. There is a variety of simple queries already defined to retrieve specific information, for example to find out what functions are actually called by a specific function.

A simple example of using xref:

- `xref:start(s).` – start the xref-server
- `xref:add_module(foo)` – add the module foo. The module must be compiled with the Erlang/OTP compiler flag `debug_info` to guarantee that all the interesting information can be retrieved.
- `xref:analyze(s, {calls, [M,F,A]})` – show all the functions that are directly called by the function described by `{M,F,A}`.
- `xref:stop(s)` – stop the server named s.
6.2 cprof

cprof [4] is a profiler module for Erlang/OTP. What it does is track the number of calls to functions (and thereby modules) in our runtime environment, making it possible to profile code and find out functions call counts (and thereby, usually, where the most time is spent). cprof uses breakpoints, similar to local trace calls, and we are therefore not forced to compile the modules with any specific compiler-flags. The information presented by cprof are the number of total calls in a specific module, and then the specific amount of calls for each called function in this module. Modules are then presented in decreasing order.

An example:

- cprof:start(). - start the cprof-server
- Execute some code
- cprof:pause(). - pause the cprof-server, making sure no new calls are registered.
- cprof:analyze(). - presents the list of called modules (and how their functions are called) in a way that is very easily parsable by Erlang.

An example of the returned information might be:

```
[[lists_sort,6047,
  {{lists_sort,merge3_2,6},923},
  {{lists_sort,merge3_1,6},879},
  {{lists_sort,split_2,5},661},
  ...
  {{lists_sort,mergel,2},79},
  {{lists_sort,rmergel,2},27}],
```

Here the module lists_sort is called a total of 6047 times, and the individual function {lists_sort,merge3_2,6} is called a total of 923 times.

6.3 fprof

fprof [5] is a profiling tool that is used to find out how time is spent in an executing program, while also showing the time spent on garbage collection and other VM-run operations. It uses tracing functions to accurately save what happens in the program to a file, using a specific syntax. With fprof it is also possible to trace directly to a specifically written tracing program. fprof provides accurate information on time spent in functions, how the functions call each other, how many calls there are to each function and the amount of time used to for VM runtime services such as garbage collections. It also provides information on the time spent in functions that are in
some way called by a specific function. If possible, fprof tries to use CPU timestamps, but wallclock-times are used if necessary.

Profiling is usually done in three steps with the fprof-profiles:

1. Trace, while executing our specific piece of code, information on how things behave are saved to a file.
2. Profile, the file is read and we generate profile data by simulating the call stack among other things.
3. Analyze, the raw profiling data is parsed into correct Erlang-syntax while still generating readable text.

In an Erlang shell this would usually, in a simple case, look like this:

1> fprof:apply(module, function, argumentlist).
2> fprof:profile().
3> fprof:analyse().

Here the function specified is run and traced, and the resulting trace is then profiled and analyzed.

This then generates output similar to the following:

```
{{undefined, 0, 1691.076, 0.030},
 {fprof,apply_start_stop,4}, 0, 1691.076, 0.030}, %
 {{foo,create_file_slow,2}, 1, 1691.046, 0.103},
  {suspend, 1, 0.000, 0.000}}.
...
```

The main function in this block is \{fprof,apply_start_stop,4\} (denoted by a %) which was called 0 times\(^7\), the time spent in functions beneath it is 1691.076 milliseconds and the actual time spent in the function is 0.030 milliseconds.

### 6.4 perfctr

Perfctr [20] is a tool for accessing a number of different hardware performance monitors (HPM) on a variety of existing platforms. The information it provides is limited by the support of the underlying platform. Most platforms provides HPMs that track information showing for instance the number of L1 cache misses, CPU clock cycles, branch-misses and other similar important performance metrics. The current versions of perfctr works well under any patched Linux-system running x86 or AMD64 processors.

\(^7\)The function is called 0 times because it belongs to the profiling module itself.
6.5 GS

GS [6] is the module of the Erlang/OTP-system that is responsible for showing graphics, providing the GUI frontend for programs requiring such things. The GS (Graphics System) is built for Erlang to provide a common API for the creation of GUIs for a number of platforms while having a backend that correctly handles the underlying graphical systems on specific platforms (X, Windows etc)\(^8\). Creating objects and receiving actions (the user pressed a button for instance) is a simple matter, but in all its simplicity, the system is limiting. A key example of this is the fact that the actual look and feel of the system is pretty much defined, it is possible to change the color and size of a button, but not the design of it.

Creating and then adding objects to the GS server are not free operations, and the time it takes will be noticeable if the user is creating a lot of objects at the same time. The choice of using GS was a simple one, it is currently the only widely supported GUI system available to Erlang/OTP.

7 Current functionality of the HiPE Tool

The HiPE Tool provides a large amount of available operations that can be used on the modules/application the user is working on. Most of them are built on the following four basic operations: compiling, profiling, timing and function dependency graphs.

With these four the tool provides advanced operations such as the batched generation of scenarios and the interactive generation of scenarios. Besides these four most basic operations, the HiPE Tool also tries to provide accurate size information and information gathered by reading the Hardware Performance Monitors (only available on specific platforms).

7.1 Visual design

The visual design of the tool has gone through a number of changes, during the development of the HiPE Tool. But the main idea has not changed, to try to keep it simple and clean as possible and to provide information to the user in the best possible manner. As the tool uses the GS module (see section 6.5) to draw information the ability to modify the look and feel is limited, and the same basic idea for showing information is used throughout the HiPE Tool.

As information is shown this is usually done by having a listbox of items where the user can pick what specific data-item interests him and then have an area of the same window that shows this information.

Most windows are resizable (albeit not very nice looking at the absurd sizes), if some information is not completely shown it is possible to just increase the size of the window.

\(^8\)Compare this to Java’s Swing implementation
What has happened to the visual design is that the initial idea was to keep it very clean, but as the design of the tool progressed it became more and more cluttered. Later decisions were then made that removed most of the duplicated efforts (i.e. having buttons and menu-items that did the same thing), removing information that was either not needed or shown somewhere else (i.e. removing labels that showed you if a function was exported even though this information is already shown in the listings of the functions in the selected module). The resizing was also something that was added quite late during development, which meant some simplification of the code (with the frames needed to be able to resize, it also removed the necessity to provide the coordinates for each object (such as labels or buttons), and instead place them directly into the grid).

The HiPE Tool is designed to open a variety of new windows to handle specific tasks, but there is however a main window where the user can add modules, compile and other basic operations. This window has two boxes that show the modules added, and as a module is selected, the functions available in this module. The user may also specify the entry point for an application and profile and/or time it. This is also the window where the menubar to access specific functions in the tool (showing information, more advanced operations etc).

Also shown in the main window is also a text area that is updated during operations to inform the user what is currently being done.

![Figure 1: The main window in the current version of the HiPE Tool.](image-url)
7.2 Adding files

The first choice the user must make is to choose exactly which modules he needs to add to the tool, in essence deciding what application he is trying to process. All the interesting modules should be added so that they can be correctly compiled, profiled etc. Quite often it is not obvious exactly what modules will be used, so the user may end up adding all the files in the application’s directory.

What then happens is not obvious, and to some users may feel quite uninventive:

1. The selected items are filtered to remove any directories or files not ending with .beam or .erl
2. The data is split up into modules and source-files (.erl), and any source-files that are added as beam-files are removed (source-files are thereby only added if we have no compiled version of it).
3. The modules and their home-directory are added to an internal list in the program
4. The files are loaded to guarantee that they can be accessed even as we change the directory later on, and the sizes of the added files are checked if we have enabled that option.

The user can then change the directory and add new files. Modules from any number of directories may be added.

Figure 2: Adding files to the tool. Note that you can either add all the files in a directory or specific files.
7.3 Profiling

Profiling is done either in a lightweight or a more heavyweight fashion. The goal of profiling is to find the hotspots of the program, i.e. the places where the most time is spent. It is therefore a necessary first step for efficiently compiling an application.

It is interesting that while we limit ourselves to the actual profiling modules supplied in the Erlang/OTP system there has been a wide variety of tools developed to in various ways profile applications, an example is gprof [14], developed in the early 80's.

7.3.1 Lightweight profiling

The lightweight version of profiling uses the cprof-profiler (see sec 6.2) to provide an accurate listing of the called functions. This is a low-overhead tool that has certain advantages and disadvantages:

+ The lightweight profiler is very fast. Retrieving the gathered information takes a few milliseconds, which means that we do not add significant amount of time after the profile is done.

+ The profiler does not create any items on the disk – all data is stored in memory, taking up minimal amount of space.

+ Has a very low impact on the performance of the system.

– The profiler does not accurately gather information on the garbage collection or other VM runtime services, these are not directly handled by the shell and will not show up in the profile.

– The profiler gathers information on ALL the activity that goes on in the shell. This includes loading files and other such things, which might end up influencing gathered data in some ways.

7.3.2 Heavyweight profiling

The heavyweight profiling is done with the fprof-profiler (see section 6.3). Using tracing and a more fine-grained selection this overcomes many of the disadvantages of the lightweight cprof-profiler, sadly it adds a few of its own. As with cprof, this can be summarized as a list of advantages and disadvantages:

+ Very fine-grained, provides accurate data on time spent, calls to specific functions etc.

+ Provides information on VM services such as garbage collection,

+ Opens the door for further examination of the gathered data.
- Impacts the performance of the execution to a far greater extent, slowing this down.

- Takes a very long time to finalize the profile, actually profiling and analyzing the gathered data can take a very long time.

- Creates a file on the disk to hold all the trace data, which may grow so huge as to go over the file-system limit. This makes the tool almost unusable on applications with a lot of calls (over a few hundred thousand).

### 7.3.3 Heavyweight or Lightweight? You decide!

With the advantages and disadvantages of the previous sections in mind, which profiling tool to use may not be an obvious choice. However, with the fprof-module having severe difficulties profiling under some circumstances, the lightweight cprof-profiler is the one enabled by default in the HiPE Tool.

The user may deem if his application can handle using the fprof-profiler (usually the extra time needed or the large number of calls in an application is the reason for not using it) and instead use it, but anyone that wants to use it should be aware that as it tries to create files holding the tracing data, any application with a very large number of calls (over a million or so) will probably not cooperate with fprof (during trials it was very common to reach the maximum allowed size of a file for the filesystem, around 4 GB, while profiling applications with large amounts of call). fprof should be left to applications with few calls, where the user is more interested in tracking the behavior of the application.

The more advanced applications of the HiPE Tool all use the cprof-profiler, mainly due to the problems of the fprof-profiler that have been explained, and the fact that currently no extra information that the fprof-profiler and not the cprof-profiler can gather is actually used in these more advanced applications.

### 7.3.4 Native code and profiling

When a function is compiled to native using the HiPE compiler any reference to this function is trapped by the runtime environment and redirected to the native version of it. However, the native code does not report accurate trace data in the way that the fprof profiler recognizes, nor will the information show up in the cprof profiler. This means that any function/module that is compiled to native code will not show up at all in the gathered profile data. Any previously compiled function is therefore de facto invisible to the supplied profilers; a fact that the more advanced functions of the tool exploits - as the hotspots that have been removed by compiling are not shown to us if the application is profiled again, it makes the next unattended hotspot very easy to find.
Figure 3: A profile done by the lightweight cprof profiler. Here we can see: what modules were used, what functions were called (and how many times) and we can investigate specific functions to see what they call. We can also visualize this using the dynamic function dependency graph for this execution.

7.4 Compiling

The HiPE Tool provides a variety of ways to compile specific functions or modules, while also using the compilation of such items to provide the more advanced functionalities inside the tool. Compiling already loaded modules is a simple matter of selecting the module you wish to compile and either pressing the HiPE-button or the BEAM-button to compile (with the HiPE- or the BEAM-compiler). This will compile the module, replace the existing version of it and then load the new version. As modules can be located from anywhere in the file system, the current directory is changed to the one where the file was added from and the compilation is done using the source-file as specified in the module_info gathered from the beam-file. This guarantees that we are compiling from the right source. As the file is reloaded one also guarantees that the shell is using the latest code available. As has been mentioned several times in this report, with the HiPE-compiler the user is free to actually compile specific functions to native code, something that can be easily done in the HiPE Tool by selecting the module and then the function the user wishes to compile and pressing the button to compile this function using HiPE. As this is done directly in the shell, working on loaded code there is no need of changing directories or reloading code. Reloading the code from the .beam-file would actually replace the new native version of the function.

Earlier versions of the tool spawned a new process that handled the actual compilation and then reported back. This is still done for the compilations that the user initializes, but not for the large batched compilations done for instance while starting the xref-

\textsuperscript{9}module_info is a automatically generated function for each module that shows the location of the sourcefile, flags used to compile etc
The HiPE compiler also has a variety of compiler options available, including various levels of optimization, register allocation algorithms etc. The HiPE Tool provides a flexible interface to these options, accessible as a menu item. In here the user can select an overall optimization level and then, if the user deems it necessary, he can go in and exactly specify the options that should be used. However, this is left to those who know exactly what they are doing. By default the o2 optimization is used, but whatever settings the user decides on are used for compiling with the HiPE compiler. Compiling a specific function up until quite recent versions was done by using the default o2 flag at all times, but in the current version functions are also compiled with the compiler options the user has selected.

![Compiler options available to the user. Most users will only want to change the o1,o2,o3-levels, but advanced users may want to fiddle with the other flags.](image)

**7.5 Timing**

Timing can be carried out in the tool itself by the click of a button. What happens then is that a hardware timer is started, the defined work is done and the timer is stopped. This hardware timer makes sure that no actual idle time is included in the timing, but
it also not guaranteed that all the time needed is spent actually executing the program the user is interested in. All the VM-specific items are also included in the timing, for instance the time spent in garbage collection. The timer has a resolution of 10 milliseconds, which is in all respect good enough for this implementation, as there is already overhead from the various runtime services. As the application is timed, the HiPE Tool also gathers data from a number of hardware performance monitors (see section 7.6).

The times gathered can then be shown to the user, who is specifying the entry point the application was started with, the time it took, and the various performance metrics that were gathered, if any. Timing an application is usually a good idea before starting to compile functions or modules, to have a good baseline for when you gather timings when some items have been compiled. Timing also plays an intricate part in the more advanced options available in the HiPE Tool, and is the only real way of measuring whether a compilation improved or worsened the situation (or as quite often happens, the compilation did very little to the overall execution time).

### 7.6 Performance Metrics

As the execution of an application is timed, the HiPE Tool also gathers data from a number of hardware performance monitors. The data gathered by these monitors is currently not actively used by the HiPE Tool, but it might be of interest for later improvements. The metrics may also be of use to users who are interested in trying to improve for instance the number of L1 cache misses of their applications.

The metrics are gathered using the perfctr-program (see section 6.4), with the aid of a specifically coded module for Erlang called vperfctr. This module has been designed to work with the AMD64-processor, making the gathering of data unavailable on the x86 at this time (even though, as written in section 6.4, perfctr can handle this platform). The vperfctr-module has a simple interface for deciding what counters to access and the current incarnation of the HiPE Tool gathers information on L1 data cache-misses (cache misses in the CPU’s own processor of data) and it calculates the CPI (Cycles Per Instructions, defined as #instructions/#clockcycles) by gathering the number of retired instructions and the number of cycles. The HiPE Tool also gathers information regarding the mispredicted branches of the execution.

Besides the metrics gathered by vperfctr, the HiPE Tool also gathers information about garbage collection of the runtime system. A heavy amount of garbage collection might indicate performance problems in our application. Even in a large commercial applications only around 5% of the total execution time is spent garbage collecting [12].

As was written, these metrics are not used by the HiPE Tool besides actually showing the gathered information, but in future implementations the tool should try to actively use them.
Figure 5: Listings the time trials done. Note the fields CPI, Cache misses etc. These are the hardware performance monitors readouts, but as this was not run on a chipset that is currently supported, no such data was gathered (they are N/A).

### 7.7 Xref-server

The Xref server is the basic server that is needed to run the xref-module (see section 6.1). This is not automatically started, and the user will need to start it, and update it as new modules are added to be processed by the HiPE Tool. What happens when the xref server is started/updated is the following:

1. If the xref server is not started it is.
2. It finds any new modules it needs to add.
3. It tries to compile these modules with the BEAM compiler using the `debug_info` flag which is needed to see all the information we are interested in. If the module is already compiled with this flag it is not compiled again.
4. The modules are then one by one added to the xref server.

The xref server should be started/updated before the user tries to do anything besides timing or compiling, almost all of the other operations demands that the server is up to date. Starting or updating the xref server takes some time. Compiling the modules to be added to use the correct compiler flags can take a while if they are large and/or there are a lot of modules to be added that aren’t correctly compiled. Adding these now compiled modules to the server isn’t a free operation either, it does take a noticeable amount of time to add a single module, and adding several hundreds of them can take a few seconds.
7.8 Investigating function dependencies

With the xref-server started the user is free to investigate specific functions in the added modules. Investigating them in this context refers to using the xref-server to find the descendants of a specific function. This gives the user a clear idea how the function behaves in the context of the other functions. The descendants given are besides the one in the added modules also functions in non-added modules (usually the modules provided by the Erlang/OTP release itself, modules of the standard library such as lists and string). But as these are not added to the tool, the functions that are called from these modules are not in turn investigated for what descendants they have inside the modules, as this information is not available when it comes to modules that have not been added.

If this investigation is done via the normal menu items the static function dependency graph is created, but if the user investigates a function in the context of a profile, the active function dependency graph is gathered. With the information gathered from the profile it is possible to filter out those functions that fall below a certain threshold (currently 0 calls), which provides this active function dependency graph which more accurately describes the reality of the execution (see section 4.4).

It is then a trivial matter to present the information gathered by the investigation to the user and allow him or her to compile the function and its descendants using a variety of options. The HiPE Tool allows the user to compile the module of the investigated function and/or all those modules with functions that are descendants to the investigated function. Besides this, the HiPE Tool also allows the user to only compile the investigated function and its descendants directly.

Figure 6: The function qsort:test/1 has been investigated. Note that the descendants of this function include a lot of function in the qsort-module itself, and a few outside of it. In the menu it is also shown that this was a static function dependency.
### 7.9 Sizes

The size of the code as it is loaded into the system is an interesting metric to consider. As Erlang is often used in embedded systems, where memory is a scarce resource, and most CPUs have a very limited cache, conserving the total size of the loaded code is an important concept. If the loaded code is too large this might degrade performance.

The following section describes the current size gathering functions for non-native code used in the HiPE Tool. However, experiments seem to show that it is returning incorrect size information. More information can be found in section 11.1.

Collecting the size of loaded modules is not a trivial matter - as the interesting metric is the size of the loaded code in the system, it is not just the simple matter of investigating the size of the modules on disk (in essence checking the size of the .beam-files). Both the native and the Erlang/OTP code-loaders does major magic when it loads in the code, unpacking it in essence, which will increase the size of the code once it is loaded. There are then two cases to consider, the size of the emulated bytecode, and the size of the native code. In the first case there are commands to disassemble the now loaded code to a file on disk, and investigating this file should give a more accurate reading as to the total size of the module as loaded by the Erlang/OTP loader. The second case, the size of the loaded native code, is instead checked by a simple built-in command in HiPE, which provides the size of the loaded code for a specific module.

Investigating the sizes of loaded bytecode is not a cheap process, as described in the previous paragraph, the file is disassembled and then the size of this disassembled file is investigated. This takes in the range of a few hundred milliseconds, which means that for a good number of modules this can take quite a while. The non-trivial times involved means that the user needs to have a better control of when this data is gathered, especially for when the user adds thousands of modules, and the tool provides a switch for deciding if the sizes should be gathered. If this switch is turned on, the size of modules are gathered as they are compiled, or added to the tool. In the more advanced scenario generation algorithms the size of a module is also needed to be able to rank the scenarios according to their time and their size.

The information on the sizes of the added modules are available in a specific window, showing the sizes for each functions in all the various ways they have been compiled, and the ratio between these and the size of the most basic emulated bytecode. Also presented is the total size of all the modules in the way they are currently compiled.

This size gathering system is currently not working as it should, this is discussed in section 11.1.
Figure 7: Listing the file sizes gathered from the currently added modules. The N/A means the program have never witnessed the module in this mode, and have therefore no data on it.

7.10 Visual call graph

The visual call graph is the visualization of the function dependency graphs (sec 4.4). The visualization can be done as the user is viewing a profile, to in a better way visualize what has happened, how the application behaves. The user is allowed to either visualize the profile starting with all the functions of a specific module (or all modules), or with a specific function, showing this function and how its descendants interact.

The graph is then generated according to the following idea:

1. The called function(s) is/are placed furthest to the left.
2. The functions these call are placed in the next column, the functions these called function call are placed in the column after that, and so on.
3. Any function that has already been placed remains in the original place (guaranteeing that each function is only placed once).
4. Directed edges are drawn showing how each function call other functions. This does not include recursive calls – that is, if a function calls itself there is no edge from the function to itself. The edge points to the function that is called.
5. Besides placing the name of the function, the HiPE Tool also places the number of calls to this function (or the string Native) inside a rectangle that represents the function.
6. Those functions whose number of calls is over a certain thresholds and whose percentage of the total calls is over a certain threshold as well are colored in different shades of green, to show the user the hotspots of the program.
The user can then click on any of the functions and thereby select it, and also see all the descendants it has, colored in deep red/light red. As the user is viewing this visualization he can decide to investigate the currently selected if he wants to, or compile them (using the options for the HiPE compiler as chosen in the HiPE Tool) using the native HiPE compiler. Either only the selected function is compiled, or the selected function and its descendants.

Built into the system is also the capability of zooming in and out on the generated function dependency graph. The available zooms are 16x, 8x, 4x, 2x, 1x - where 16x is the default value. Only in 16x and 8x is any text written, in the more zoomed out modes any text written would be unreadable.

The user is also presented with the option of saving the generation call graph to disk. This is done by using an Erlang interface to the graph language DOT [2], and then using the generated DOT-code to create either postscript or GIF using the software dotty [1]. As dotty is a specialized piece of software designed specifically to draw graphs in a clear and working manner from DOT-files, the graphs generated by it and saved to disk looks very dissimilar to those presented in the HiPE Tool itself.

7.10.1 Problems with the visual call graph

As always, if a large number of processes are being processed, the time for execution increases. In an example with the HiPE compiler involving some 1800 functions and 4400 different pairs of caller/callee functions, the process of creating and placing all functions for the visual presentation of the function dependency graph took over a minute on a slow machine. However, the necessity of using these sort of call-graphs for such huge function dependency graphs can be debated, with 1800 functions, the overview of everything suffers to great extent and it might be debatable if it provides the user with a usable readout. With smaller graphs, such as the one for the qsort-algorithm shown in Figure 8, the reason why this is supplied in the HiPE Tool becomes evident. This is a very clean, very intuitive way of investigating an application/module.

7.10.2 Deciding on thresholds

The thresholds, used to determine if something is a hotspot for coloring purposes, are in the current version of the HiPE Tool hardcoded into the program. The thresholds are: 300 calls and a minimum of 10% of the total calls of the shown functions. These thresholds are merely guesses as to what might be a good way of only coloring those functions which are de facto real hotspots.

The reason there is a minimum number of calls is that in any call-graph with a very few number of total call, getting over a percentage of the total calls is very simple, a single call might be enough. In the same way, if there are a lot of calls, and they are spread out over a large number of functions, several functions might end up having more than a specified hard limit on the number of calls. This might not be interesting if you compare it to the total number of calls, showing why a percentage of the total
calls is needed in conjunction with just a direct counting of the function’s calls.

Figure 8: Visualization of a profile of the qsort module with zoom factor 8x.

Figure 9: GIF image corresponding to the visualization shown in Figure 8
8 Advanced Functionality of the HiPE Tool

Most of what has been described in the previous sections are simple building blocks of the tool (compilation, profiling, timing, dependency investigation etc). Common sense tells us that the user will want to automate many of these things if possible. By looking at the situation, it is obvious that a good algorithm for finding a close to optimal way of compiling an application is by first profiling it, compiling the portions where a lot of time is spent, and then timing it to see what happened (hopefully decreasing the execution time), and then repeating the process until all the hotspots are gone. The two implemented advanced operations of the HiPE Tool try to automate this procedure.

A key idea for these advanced methods is how you can find hotspots in Erlang programs. As Erlang is a functional language without loops, and instead uses recursion, any function where a lot of time is spent is called a great number of times. These are almost always recursive functions. After finding these heavily called functions it is also a good idea to compile their descendants, to guarantee that we have as few mode switches as possible.

The HiPE Tool tries to remove the most active hotspots whenever possible, in essence using a greedy algorithm to compile the hotspots that represent the most amount of calls in each iteration.

8.1 Interactive scenario generation

The interactive generation of compilation scenarios is one of the two ways of creating compilation scenarios in a controlled fashion. This interactive generation tries to automate the procedure described previously, but to a lesser degree than the batched generation detailed in 8.2. The user provides the entry point to his or her application, the interactive generator then tries to iteratively find better and better scenarios by compiling more and more functions to native code in an effort of increasing efficiency and decreasing the time spent executing.

After being supplied the entry point the HiPE Tool enters the interactive, iterative loop, which does the following:

1. **Profiling** – the entry point is run, while it is being profiled using the cprof lightweight profiler. The HiPE Tool then finds the most called function, the largest hotspot of the program.

2. **Compiling** – the hotspot gathered by the profiler is then compiled along with its descendants to native code.

3. **Timing** – The application is timed to see if there was an improvement from the earlier scenario generated, or even if things are now worse.

4. **Decision** – In this step the user is able to decide if he wants to start over with the loop and create another scenario or if he is satisfied with the current scenarios.
This is guided by the tool, it tries to give hints as to what the user should do, by itself comparing the current and previous timings. If the user is satisfied he is taken to a window where he can decide which scenarios was of most interest and these can then be saved to disk.

The interactive generation heavily uses the components earlier described, it uses the easy access to the native compiler, the lightweight cprof-profiler and the timing mechanisms that are in place.

The interactive scenario generation also exploits the fact that no native code will actually show up in our profile, guaranteeing that once we’ve gotten rid of a specific hotspot, it will never again show up as a hotspot, and the next heaviest hotspot will be compiled. The interactive generator always pick the most called function as the current hotspot to compile, along with its descendants, effectively trying to cut away as many of the calls as possible with each compilation.

![Interactive scenario generation](image)

Figure 10: Deciding if to continue or not during the interactive scenario generation.

### 8.2 Batched scenario generation

The batched generation of detailed compilation scenarios iteratively creates scenarios. The name comes from the fact that it creates scenarios in batches, ending after a stopping criteria has been reached. The user can first choose between a small group of these criteria and then select the entry point for the application to be processed.
The user then lets the automated batched generation-process generate compilation scenarios according to what rules were picked. After completion the user can select which of the generated scenarios worked best, with some help from the tool.

The stopping criteria, or rules, available to the user are:

- Minimum percentage time gain between compilation scenarios. E.g if scenario #1 had a time of 100 ms, scenario #2 must have a max time of $0.9100 = 90$ ms, or else the batched generation halts.

- A maximum time to spend generating scenarios, in milliseconds. Before trying to start a new scenario, the HiPE Tool investigates the time spent generating scenarios, if this is more than the maximum allowed time, the generation is halted.

- Maximum amount of scenarios to create. A simple upper limit on the number of scenarios that should be created.

Only one of these options can be chosen by the user at one time.

Figure 11: Selecting rules and the entry point for the batched scenario generation. The JiT-box is discussed in section 14.1.

The generation of a scenario is done in the following fashion. The application is profiled (using the lightweight cprof-profiler) and timed, the largest hotspot found, and its descendants, are then compiled and a scenario is generated. This generated scenario is the scenario before the HiPE Tool actually compiled the current hotspot - this is done to be able to connect the current timing to the correct scenario. The criteria used to halt the execution is evaluated, if the criteria has not been reached, the process starts over, to try to find new hotspots to compile. If the user is trying to generate a set number of scenarios, has very low demands on the time gains needed or has picked a
very long maximum time to spend generating scenarios, it is very possible for the tool to run out functions to actually compile. What then happens is that the HiPE Tool applies a set of heuristics, picking a more advanced ("heavier") selection of compiler flags, and then starts compiling the already compiled functions with these new flags, by the order in which they were originally compiled. This procedure is then repeated until either the criteria is reached, or all the available functions have been compiled with the heaviest flags possible – which means nothing more can be done.

After the scenarios are created and the process is finished, the user can select those scenarios he finds most interesting, i.e., those that either spent the least amount of time, or those that spent the least amount of time without going over a set limit on the size of the native code (or the number of compiled functions).

![Figure 12: Running the batched scenario generator.](image)

### 8.3 Which advanced mode to use?

When should the interactive generation be used, and not the more automated batched generation described in section 8.2? There are a few times when it should be used, one might be when the user is very unsure what kind of rules would be best to use for the batched generation (i.e he can’t come up with any rules he thinks would do a good job). Another is when the application that is being processed involves a GUI, meaning the user himself must decide what to do in the application. The reason here is quite simple, the batched generation is more of a fire and forget mechanism, but in the interactive generation the user, in each loop have the possibility of trying out a different aspect of the items in the GUI. Still, one can use both of these methods quite interchangeably, just that in the batched generation the rules for when we actually halt is very much defined, in the interactive case it is much more fluid, it can be stopped
after just a few loops if the user wants to, and in simpler applications this is often the case.

8.4 Selecting profiles

The selection of profiles occurs after the user has correctly finished either the interactive scenario generation or the batched scenario generation. The goal of selecting the scenario is to let the user decide which of the generated scenarios does what he felt was the best job, this is not always the best time, the user might very well want the best time gain possible, but with as few functions in native code as possible. For a great number of applications the following apply: after compiling quite a few functions (and their descendants), compiling more functions to native code does not effect the execution time in any great way. This usually happens if there are a small set of functions that does almost all of the heavy work involved, and there are a lot of peripheral functions that are called quite seldom. So the user is presented with the list of generated scenarios, initially sorted by their time; sorting them by the ratio between the number of compiled functions and the time the execution took is also possible. The user can then save those scenarios which seemed to be the most interesting for later.

The scenarios that have been selected from successful runs of the interactive/batched scenario generation can then later on be accessed and saved to disk. A correctly formatted .erl-file, containing the directives to the native compiler for compiling our chosen detailed scenario is created and saved. As the source code does not specify where the modules that will have functions compiled are located, the user is forced to in some way make sure that the modules are available for compilation. This can be accomplished either by making sure all the modules are in the current directory where the native compiler will look, or that they are already loaded. An example of a created .erl-file is shown below:

```erlang
%% COMP-FILE GENERATED BY HiPE TOOL v0.9666
%% Created on 2004-12-20
%% At 17:23
%%---------------------------------------------
-module(saved_scenario).
-export([compile_all/0]).

compile_all() ->
    catch hipe:c({qsort,partition,5},[o2]),
    catch hipe:c({qsort,qsort,2},[o2]),
    catch hipe:c({qsort,mkrandlist,2},[o2]),
    catch hipe:c({random,uniform,1},[o2]),
    catch hipe:c({qsort,loop,3},[o2]),
    catch hipe:c({qsort,qsort,1},[o2]),
    ok.
```
It could be possible to automatically create a process where a profile is selected from
the list of created scenarios according to a specific rule-set designed by the user. You
could for instance create a scheme that would weigh the time/memory constraints of
the application together and then would try to select the best possible choice. As this
has not been explored fully, the HiPE Tool simply allows the user to manually select
those scenarios that seem interesting.

9 Changed or removed functionality

The following items are operations of the tool that were once apart of the tool’s design,
but have been removed due to either not being used, or there not being a good way
of actually implementing the ideas. Also included are some entries where the basic
implementation of the operation have been changed in a non-trivial manner.

9.1 Scripts

In versions 0.6 through 0.7 there user had the possibility of creating scripts for the
HiPE Tool. These scripts could automate the more basic operations of the tool, such as
changing directories, compiling modules/functions, profile, time etc. The main reason
why they were removed was due to the fact that the user had to sit down and create
these scripts himself, when he very often only wanted to do the same basic operations
over and over. So instead of relying on the user to create the scripts, the more advanced
methods of the interactive scenario generator and the batched scenario generator has
been implemented, replacing the scripts. In the current version there is no reference
to scripts at all.

9.2 eprof

The profiling tool eprof was implemented and used in the tool from the very start to
up to version 0.6. This tool is faster and creates less overhead than fprof (see section
6.3). What it does is find out the percentage of the total time that was spent in
specific functions, which could help us find the hotspots of the program. However, the
possibility of using this tool has been removed, and instead the cprof profiler is used,
which is even faster and gives about the same functionality.

9.3 Shell-scripts

Some applications are started with normal bash shell–scripts in a UNIX-environment.
The HiPE Tool tried to allow the user to use these scripts to start his or her application,
instead of having to actually pick an entry point for the application. The problem that
finally forced this possibility to be removed was that there was never a good way of
starting these scripts in such a way that you could actually profile what was executed.
What the HiPE Tool did to use these scripts was to call the OS and have that execute the script. The problem here was then that the script usually started its own runtime environment, in such a fashion that it was very hard (if not impossible) to ”contact” this environment and be able to actually profile what was done in it. The current version of the tool has removed any references to these shell-scripts, the user is now forced to supply an entry point to the application he or she wants to process.

9.4 Adding files

The act of adding files that is currently used is described in section 7.2, but this has been changed as the tool has been revised. In the earlier versions of the HiPE Tool the user was only allowed to select the entire content of one specific directory, making it impossible to select exactly what we wanted. The upshot of this was that the HiPE Tool always knew what directory the user was interested in, making sure it didn’t need to be changed. This also allowed the HiPE Tool not to be able to add files from other directories, nor was it necessary to make sure modules were loaded. In the currently released version however the HiPE Tool (as written in 7.2) allow the user to add source files and be very specific what modules he wants added, giving greater control, but demands more of the tool (changing directory, ensure modules are loaded, etc).

9.5 Gathering sizes

As is described in section 7.9, gathering sizes is not a simple matter of reading the size of the original .beam-file on disk, but one needs to disassemble the file, and then investigate the size of the disassembled file. However, due to the handling of file pointers it was very possible to go over the max number of open filepointers allowed for the application as you tried to retrieve the sizes. What is now done is that a new process is spawned, which is responsible for disassembling the file, and retrieving the size of it (and then deleting the generated file). After doing this, the new process returns the size of the file and is then killed, together with all the open filepointers it had.

10 Comparison to similar tools

There is an incredible amount of tools and programs that do similar things to that of the HiPE Tool (i.e improving the performance of code executing in virtual machines or investigating function dependencies and how the program executes by various forms of profiling). A broad stroke can be made to divide these into two wide categories. First we have the tools that essentially try to profile the application in some fashion, to either improve performance or to study it. Then there are the ones that try to improve the performance of virtual machines by finding and native compiling hotspots in a JIT fashion. This kind of compilation is not as simple as one may think, there are
tradeoffs in how much time you can spend on compilation vs the time you may gain from compiling (i.e only compiling the true hotspots), and these sort of estimations is not a trivial matter \[16, 21\].

The HiPE Tool falls between these, it is designed to try to find the best performance possible in what can be seen as offline operations and then use this to essentially do the just-in-time compilation before starting to execute the program.

We can of course compare the HiPE Tool to a small group of interesting tools that fall into the two mentioned categories to find similarities and differences, and to find essential ideas that might be interesting to incorporate in future versions of the HiPE Tool.

10.1 VTune

10.1.1 Description

VTune is a tool developed by Intel to ”streamline your code in just a few clicks.” \[18\]. Among its key features is the ability to what is currently being done by the system by using system wide sampling during execution. Sampling means that in some specific time intervals the program investigates what is being done. As this is done quite seldom, the performance impact is minimal. This also means that the code does not need to be changed in any way, and that you can quite easily find the hotspots of the program.

VTune also supplies the user with the ability to use call graph analysis, detailing how threads are created, which functions are calling which and so on. An interface to using a variety of performance counters is also provided.

10.1.2 Performance and experiences

As with any commercial product, getting truly independant scientific studies done is quite rare, and VTune is no exception. However, VTune itself isn’t truly used to speed up the code, it merely highlights the bottlenecks and provides other data, gathered mostly by sampling. Standard usage by normal users seem to indicate that this works as well as expected.

10.1.3 Comparison

It is obvious from a mere glance at VTune that it shares a lot of ideas in common with the HiPE Tool. Good examples of this is the investigation of the programs hotspots and the availability of function dependency graphs to visually overview the program. The use of sampling employed by VTune is however more powerful than our tool’s way of simply investigating the total number of calls to the functions. Using the idea of sampling you can at each sampling instance not only gather what is currently executing (gathering a more accurate image of where time is spent than simple number of calls done) but also the current status of hardware performance monitors.
VTune is primarily interested in demonstrating the performance of your code, and show you where time should be spent improving it, it doesn’t in any way dabble with compilers.

10.2 Performance Explorer

10.2.1 Description

Performance Explorer is a tool developed to investigate trace data gathered from executing Java applications in a custom-made Java Virtual Machine. [23]

The tool works with two concepts; trace records sets and metrics. The trace records sets are a collection of trace records. A trace record contains interesting information, primarily of what is is being done, and what is the status of hardware performance monitors. These records can then be filtered in a variety of way, for instance by only allowing those that contain a specific amount of cache misses, or only those ending in a specific Java method.

The metrics in turn are specific values extracted or computed from the trace records. Examples might include CPI, cache misses or branches predicted incorrectly. Some are either directly gathered from the chips hardware counters, while others are calculated from other gathered metrics.

Using these two key concepts the tool then tries to show the user what exactly is going on, by allowing him/her to filter through the gathered data and pick out interesting information.

10.2.2 Performance and experiences

Using the Hardware Performance Monitors and the data gathered from them, Performance Explorer seems to provide accurate, interesting and moldable data. Moldable in the sense that the data gathered can be used to create several different user-defined metrics, as different users have different goals. The GUI of the program might need an overhaul though, as in the incarnation presented in the paper is overly cluttered with buttons and data.

10.2.3 Comparison

Performance Explorer is quite different from the HiPE Tool. It primarily an investigatory tool, as was VTune, but its main interests are the hardware performance monitors provided by the chip manufacturers. Although it’s highly specialized it does provide interesting ideas for future developments of the HiPE Tool, in the current HiPE Tool performance monitors are integrated on very specific platforms, but they aren’t used in a very interesting fashion. What the Performance Explorer does well is to use these monitors in conjunction with execution data to provide poignant information to the user, and hopefully something similar can be used in the HiPE Tool in the future.
10.3 Java HotSpot

10.3.1 Description

Java HotSpot is a custom-made Virtual Machine for Java, designed by Sun [3]. It’s chiefly designed to improve the user’s experience in running Java programs by increasing performance. Among the chief advantages of HotSpot over Sun’s original virtual machine is: a better garbage collector, stack improvements, thread synchronization improvements and the one most interesting for this report; optimizing the JiT compiler and finding hotspots. The discussion will be mostly limited to the last advantage mentioned.

What usually happens when the user runs a JiT-compiler is that before each function is run in the virtual machine, it is compiled to the native system that is running the program. The problem with this is that a waiting cost is incurred as the virtual machine is compiling. What is done in Java HotSpot is instead that the program is allowed to run for a while in interpreted mode, and during this time it analyzes the execution, trying to find the hotspots of the program. It then tries to focus its attention on these specific hotspots while ignoring the other functions involved, making it possible to better run a full-featured compiler effort (i.e doing all forms of normal compiler optimizations which would usually be ignored in a simple JiT-compiler system).

10.3.2 Performance and experiences

The Java Hotspot VM seems to be able to speedup the execution of your standard code, using more advanced ideas to provide native code and other interesting attempts at achieving general speedup and throughput. Various tests done indicates that the Java HotSpot VM performs quite a lot better than the previous incarnation of the VM, and also beats other more advanced ones.

10.3.3 Comparison

While Java HotSpot is a far more overreaching attempt at running code as efficient as possible the ideas of trying to improve just those areas of code that are mostly responsible for the execution time (i.e the hotspots) are mirrored in what the HiPE Tool is trying to accomplish. The HiPE Tool does this in an offline fashion (i.e the investigation of what to compile etc is done before actual live use of the application), while as Java HotSpot is a full working virtual machine which does the optimization during runtime.

10.4 Other tools

Excluding the above mentioned three tools there has been a veritable plethora of work done about optimization/profiling. Two worth mentioning are: Digital Continu-
ous Profiling Infrastructure [8], an infrastructure designed to do longtime continuous sample-based profiling of a system and Dynamo [10] which gathers data from a stream of native code and transparently improves performance by creating optimized versions of very active "instruction sequences" that are run when appropriate.

11 Bugs and Issues

In any application, there will be bugs, and the HiPE Tool is no exception. If possible, most bugs are removed during the development, but some usually manifest themselves during later trials. The bug shown here is obviously known, and not yet corrected.

There are also issues that show up but haven’t really been looked at.

11.1 Size gathering

The size gathering functions used in the HiPE Tool, described in section 7.9, is currently not working correctly. The HiPE system has a built in function that returns the size of the native code for a specific module. Currently this size is far lower than what is reported via the normal disassembling used to gather the size of the non-native code in our module. This leads us to believe that the described method for gathering non-native code size is incorrect, however, no other method has been described.

11.2 Compiling functions outside the added modules

When one is using one of the more advanced functionalities of the HiPE Tool it is quite possible to end up in a position where functions are compiled from modules that aren’t actively investigated. A good example is the qsort module described in section 12.1. As it executes it calls two outside modules, one user-created and one provided in Erlang/OTP. As this module is run through the batched scenario generation it is very possible that those functions in these two outside modules are compiled to native code (usually due to them being descendants of more heavily called internal functions).

The problem lies in the fact that currently there is no mechanism for returning to the previous state before a compilation trial if the functions compiled aren’t in modules actively being investigated, that is, the HiPE Tool does not keep track of the functions in outside modules that it has compiled and doesn’t try to return these to emulated code for future optimization attempts.

The most damaging problem from this is, if we have during the current execution in the runtime environment compiled a heavily used function, such as those built into the Erlang/OTP system, we may not at all be informed that these functions are compiled when we start to optimize our program, in fact, they won’t even show up in our profilers due to the problems described in section 7.3.4. This means that the user may have a false sense of actual execution time, as he is helped by having built in
functions already compiled. As the scenario is then created the built in functions are
not specified as compiled, even though they contributed to the good speeds attained
during investigation.

This should be solvable by keeping track of whatever is compiled, and then include
some way of returning to the original state.

12 Results

12.1 Using HiPE Tool on qsort

12.1.1 Description

The qsort module is an Erlang implementation of the famous quicksort algorithm. It
tries to sort a randomly generated list using this algorithm, only calling two exter-
nal modules, bm that does timings, and random (a module provided by OTP) that
randomizes the list. qsort was originally designed as a benchmark program and the
entrancepoint that is used is qsort:test() that tells the program to sort the same list of
45000 randomized elements 30 times.

12.1.2 Starting point

A simple look at the profile of running the qsort module shows us that the execution of
this module involves very few modules, with a very small subset being responsible for
almost all the calls. This shows us that running this module in the HiPE Tool should
be quite successful.

<table>
<thead>
<tr>
<th>Function</th>
<th>Calls</th>
<th>(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>qsort:partition/5</td>
<td>25409040</td>
<td>90.1%</td>
</tr>
<tr>
<td>qsort:qsort/2</td>
<td>2700030</td>
<td>9.6%</td>
</tr>
<tr>
<td>qsort:mkrandlist/2</td>
<td>45001</td>
<td>0.16</td>
</tr>
<tr>
<td>random:uniform/1</td>
<td>45000</td>
<td>0.16</td>
</tr>
<tr>
<td>qsort:loop/3</td>
<td>31</td>
<td>0%</td>
</tr>
<tr>
<td>qsort:qsort/1</td>
<td>30</td>
<td>0%</td>
</tr>
<tr>
<td>random:seed/0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>qsort:mkrandlist/1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>bm:time_since/1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>bm:time_now/1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>qsort:test/1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>qsort:args/1</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td>qsort:test/0</td>
<td>1</td>
<td>0%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>28199139</strong></td>
<td><strong>100%</strong></td>
</tr>
</tbody>
</table>

Table 1: Shwon are the functions, the total number of calls they each have and the
percentage compared to the total.
By doing several timings and choosing the one that took the shortest amount of time (to exclude delays due to VM-related activity etc) shows us that that the execution of qsort:test() took approximately 14000 ms.

12.1.3 Using the interactive generator

After timing and profiling the execution of the module the first trial of trying to improve the efficiency was done with the interactive scenario generator.

The entry point was specified, and the execution was exited as soon as no important improvements in the total execution time were shown (when this happens the interactive generator informs us that ending the generation is probably a good idea).

<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Time: 13980 ms</th>
<th>New functions compiled: None - initial scenario</th>
</tr>
</thead>
<tbody>
<tr>
<td>Scenario 2</td>
<td>Time: 5300 ms</td>
<td>New functions compiled: qsort:partition/5</td>
</tr>
<tr>
<td></td>
<td></td>
<td>qsort:qsort/2 (child)</td>
</tr>
<tr>
<td>Scenario 3</td>
<td>Time: 5430 ms</td>
<td>New functions compiled: qsort:mkrandlist/2</td>
</tr>
<tr>
<td></td>
<td></td>
<td>random:uniform/1 (child)</td>
</tr>
</tbody>
</table>

Table 2: The scenarios generated by the interactive scenario generator.

As shown in Table 2, the second scenario generated was a significant improvement over the first one, but the third actually took longer than the second one to execute and the interactive generation was halted after it due to the HiPE Tool’s recommendations.

12.1.4 Using the batched generator

The other advanced method involves using the batched scenario generator. As described in Section 8.2 this generates a number of scenarios using a few simple rules.

First the rule that demanded a 10% time increase in each new scenario was used, giving us the exact same scenarios that were generated in the interactive scenario generator, as the third scenario generated simply did not improve the total execution time.

The next trial involved using the rule where we specified a max number of scenarios to create, due to the low number of functions involved the max amount of scenarios was set to 10 and the batched generation was then executed.
As we can see, there was some fluctuations in the execution-time as more and more functions were compiled to generate scenarios, but the fluctuations between scenarios is probably far better explained by VM-related activities such as garbage collection and other things than actual speedup of the code. This is especially true with such a large and complex program as the HiPE Tool running in the background. The second scenario generated, when the first hotspot has been compiled has the functions qsort:partition/5 and qsort:qsort/2 compiled, if we look at Table 1 we see that this represents around 99.7% of all calls during the execution, and is the only true hotspot.

As a final trial, compiling the entire qsort-module to native code gives a final time around 5400 ms, which is line with the times in all scenarios but the first one (i.e the functions responsible for almost all calls have been compiled).

### 12.2 Using the HiPE Tool on the HiPE compiler

#### 12.2.1 Description

The HiPE compiler is the actual compiler we are using in this program. This does not stop us from trying to optimize it, as all the files are loaded as emulated bytecode. What we are trying to run here is compiling a module to native code in the runtime environment (i.e nothing is saved to disk). The module that is being compiled is dialyzer.erl, the code for the main user interface of the Dialyzer [17], previously mentioned in section 2.4. The entry point was therefore hipe:c(dialyzer).

#### 12.2.2 Starting point

The first thing done is to time the compilation when nothing is compiled to native code. Several trials were made and the best time gathered was around 20400 ms, quite
a long time due to the complexity of the module that is compiled.

Initial profiling done on the execution showed information that leads one to believe that the scenario generation will be somewhat successful but nothing compared to the success shown in the qsort case. The three largest hotspots are all found in the same module, however, they are not each other’s descendants. The most called functions are shown in the table below:

<table>
<thead>
<tr>
<th>Function</th>
<th>Calls</th>
<th>(%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>gb_trees:lookup_1/2</td>
<td>3719167</td>
<td>12.7%</td>
</tr>
<tr>
<td>gb_trees:get_1/1</td>
<td>1393822</td>
<td>4.76%</td>
</tr>
<tr>
<td>gb_trees:update_1/3</td>
<td>1039153</td>
<td>3.55%</td>
</tr>
<tr>
<td>ordsets:union/2</td>
<td>843284</td>
<td>2.88%</td>
</tr>
<tr>
<td>hipe_sparc:type/1</td>
<td>807232</td>
<td>2.76%</td>
</tr>
<tr>
<td>gb_trees:lookup/2</td>
<td>737545</td>
<td>2.52%</td>
</tr>
<tr>
<td>ordsets:subtract/2</td>
<td>697908</td>
<td>2.38%</td>
</tr>
<tr>
<td>gb_trees:is_defined_1/2</td>
<td>644177</td>
<td>2.20%</td>
</tr>
<tr>
<td>gb_trees:insert_1/4</td>
<td>590924</td>
<td>2.02%</td>
</tr>
<tr>
<td>hipe_sparc:remove_immediates/1</td>
<td>513304</td>
<td>1.75%</td>
</tr>
<tr>
<td>hipe_sparc:is_imm/1</td>
<td>297990</td>
<td>1.02%</td>
</tr>
<tr>
<td>lists:usort/1</td>
<td>297248</td>
<td>1.02%</td>
</tr>
<tr>
<td>hipe:c/1</td>
<td>1</td>
<td>0.00%</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>29267437</td>
<td>100%</td>
</tr>
</tbody>
</table>

Table 4: Table showing the functions involved, the total number of calls they each have and the percentage compared to the total. Not all of the well over a thousand functions involved are shown.

As we see in Table 4, there are a lot of functions involved, and even though a lot of them are in the same module, they are not descendants of each other in these modules, and can be seen as separate.

12.2.3 Using the interactive generator

Once again, as described in Section 12.1.3, the interactive scenario generator is used, ending once the HiPE Tool think its a good idea to end (i.e when there’s too little time difference between two scenarios).

The following scenarios were created:
<table>
<thead>
<tr>
<th>Scenario 1</th>
<th>Scenario 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Time:</td>
<td>21120 ms</td>
</tr>
<tr>
<td>Functions compiled:</td>
<td>None - initial scenario</td>
</tr>
</tbody>
</table>

Table 5: The scenarios generated by the interactive scenario generator.

The first hotspot compiled did too little to improve the total execution time of the program, and the interactive generation was halted due to the HiPE Tool’s quite harsh recommendation.

12.2.4 Using the batched generator

After this the batched generator was run to see if there were improvements if we created a large amount of scenarios (and thereby compiled a large number of functions to native code). Using the same entry point, the batched generator was started with the rule to create 30 scenarios.

![Figure 13: The execution time for the various scenarios.](image)

As can be seen in Figure 13, the total time starts out very high, we gain some time from the first few scenarios created and then we just see the normal fluctuations of the execution, the new scenarios created are not influencing the execution time in any greater sense.
13 Discussion

The results shown in section 12 demonstrate the key ideas of the HiPE Tool, the compilation of those functions (and the descendants of these functions) that represent the most calls in the execution is a valid one. This is not surprising, the same idea is found in most of the literature, the hotspots are always where most of the effort is concentrated, for the simple reason that this is where we have the most to gain.

The attempted improvement of the HiPE-compiler itself demonstrates a weakness in this idea, the problem where there are quite a lot of hotspots, quite well spread out, is not a simple one, and the current idea has some difficulties improving the time spent, although it should be possible in most cases to decrease the time spent to some amount.

On the more extreme end, trying to improve an application where 99% of all calls occur in two functions that are also directly calling each other is a very simple matter, which the HiPE Tool does quickly and well. What was most interesting here was the ease in which this was achieved.

These results also seems to indicate that the interactive scenario generator isn’t at all that different from the batched generation; the scenarios found are the same when used on simple executions not involving any sort of GUI, as they are both doing about the same work, this was shown in both of the trials. This does lead one to question the existence the interactive scenario generation, but as the trials show, as we aborted the interactive scenario generation (on the HiPE Tool’s recommendation) we had created scenarios that were quite good already, which might be enough in a lot of cases.

It is also interesting to note the quite large fluctuations in execution-time for the applications; most of the time this was not at all due to our compilations but simple variations due to CPU load, garbage collection etc.

14 Future improvements

No program is ever complete, and this is also true for the HiPE Tool. Besides fixes here and there there are a few large projects that should be undertaken to make the HiPE Tool even better than it already is. Here, some ideas for the future are discussed.
14.1 Usage of JiT capabilities of HiPE

HiPE is a JiT-capable compiler, during runtime the user may compile modules or functions and these are loaded and used for future executions. The current HiPE Tool does not really exploit this to the full extent, the compilations done are not done during actual runtime of the applications we are looking at, they are merely compilations done for the next actual timing or profiling of the application.

A small simple module called hipe_jit had earlier been created that to some extent tried to utilize the JiT-capabilities. What it does is that it tracks the calls to all the involved modules in the execution and when these calls go over a hard limit the module is compiled to native mode (any modules with a lot of calls is probably doing a lot of work and should be compiled), to some extent trying to remove the hotspots the same way the HiPE Tool does.

The idea is to use this module in conjunction with the batched scenario generator; the user can specify the hard-limits the hipe_jit module uses, the application is then run and we investigate what exactly was compiled with these limits. The creation of these modules would be outside of the rules of the normal batched generation and should generate other interesting scenarios then that the batched generator itself creates. The current hipe_jit implementation does not allow user-defined thresholds, however, this can easily be fixed and the module then included into the HiPE Tool. The GUI for the batched generator already implements these things to some degree, as shown in Figure 11.

14.2 Far better usage of hardware performance monitors

As has been written in numerous places in this report, hardware performance monitors are vastly underused in the HiPE Tool, the usage of them is strictly limited to some simple information as the user times his application. This can of course be improved by a quite simple idea; via sampling during the execution of the application we not only get current status of performance monitors, but also of what is currently being executed. This data is saved and can then be investigated using some sort of data-explorer. The basis for this idea can be found quite prominently in Performance Explorer (section 10.2) and to some extent VTune (section 10.1).

A mock-up GUI for this idea has been created and is shown below:
Figure 14: The settings the user can select before starting a timing that will, by sampling, gather data on hardware performance monitors. The user should be able to select what monitors he wants to use (depending on the hardware) and what resolution the sampler should use (time between samples).

Figure 15: Investigating gathered sampling-data. The user can select what timers interests him, and can then narrowly select a point in time and get information on what was executed in this period and what happened to the monitors.
15 Summary

This report describes the creation of the HiPE Tool, designed to use the HiPE JiT compiler for the Erlang language in an efficient manner, to gain the most from compiling the users program and to let the user investigate the modules he has created. Using HiPE's ability to compile single functions, the HiPE Tool profiles an application and then tries to remove heavily called functions (hotspots) and those functions they talk to. Included is also a mechanism that tries to automate this idea, using a greedy algorithm of removing the largest hotspot each iteration.

The HiPE Tool was run on two separate applications and showed to be effective in applications with few large hotspots, but not very good in applications with more and well-spread out hotspots and a larger number of modules involved.

The HiPE Tool also provides investigatory functions to view size of the code, how functions interact, show module information etc.

Comparing the HiPE Tool to other applications within the same area showed it to be quite different; it straddled a border between programs that mostly investigated execution to present to the the user and fully fledged automated JiT compilers for virtual machines that tried to find hotspots and compile them as the application was running.

The HiPE Tool is currently in a usable state, but may not be as efficient or as nice to use as one would hope. It will be made available on the HiPE team website.

16 Acknowledgements

The author wishes to thank Kostis Sagonas that supervised this project, the HiPE-group at Uppsala University and members of the Ericsson Erlang team.

References


