View metadata, citation and similar papers at core.ac.uk

brought to you by CORE

provided by University of Essex Research Repository

Fine-grained Energy/Power Instrumentation for Software-level Efficiency Optimization

David J Greaves*, Miloš Puzović†, Ali Mustafa Zaidi*‡, Klaus McDonald-Maier§¶, Andrew Hopkins§

*Computer Laboratory, University of Cambridge

†MathWorks

‡ARM Holdings Ltd

§UltraSoC Technologies Ltd

¶University of Essex

***Abstract*—In the pursuit of both increased energy-efficiency, as well as high-performance, architects are constructing increasingly complex Systems-on-Chip with a variety of processor cores and DMA controllers. This complexity makes software implementation and optimization difficult, particularly when multiple independent applications may be running concurrently on such a heterogeneous platform. In order to take full advantage of the underlying system, increased visibility into the interaction between the software and hardware is needed. This paper proposes on-line and off-line fine-grained instrumentation of SoC components in hardware (e.g. as part of the debug & trace infrastructure) in order to enable improvements and optimization for energy efficiency to be undertaken at higher levels of abstraction, i.e. the programmer and runtime scheduler. Energy counters are incorporated for each component that keep track of energy use. These counters are indexed by *customer number* tags, that are used to distinguish between the transactions executed on any given component by client applications running in a multitasking SoC environment. The contents of the counters for each augmented component, correlated with the appropriate consumer-numbers, are extracted from a running SoC under test via existing debug & trace interfaces like GDBserver, JTAG and various proprietary trace probes. In addition, auxiliary processing on-chip computes local and global energy figures and offers them through a 4-layer abstraction stack so that programmer-level fine-grained energy measurement is made available. Both the O/S scheduler and programmers can adapt their policies and coding styles for their desired energy/performance tradeoff.**

Keywords: Power monitoring, debug and trace, on-chip debug support, energy-efficient computing, spEEDO.

I. Introduction

Power-efficient design of computer hardware and software is very important: the *Dark Silicon* issue arising from the end of Dennard Scaling means that further performance scaling must be tied to energy efficiency improvements [1]. Many significant efficiency improvements can only be achieved at the higher levels of abstraction, such as in choice of data structures. However, to enable programmers and O/S developers to optimize for efficiency, it is important to provide fine-grained visibility into the interaction of software with hardware components in the target system. This must be in the presence of other, unrelated applications that may also be utilizing the same H/W resources in a multitasking environment.

The extent to which run time and energy use are correlated for many benchmarks was explored by Pallister [2]. As one would expect, these are often correlated but not always. Hence simply using execution time as a guide for energy saving does not always work. Discrepancies arise from:

1) A complex CPU such as the the Cortex-A8 will compute faster but use more total energy than a simple 3-stage pipeline. (This is the motivation for ARM's Big.LITTLE).
2) Received DVFS wisdom teaches to run at a low voltage and compute slowly to get a cubic energy saving, but this may not apply to DRAM controllers or other shared resources that will be turned on for longer under this approach.
3) Turning on and using several cores at once and finishing early typically also saves energy use in shared caches.
4) Use of coprocessors and SIMD units by the compiler may save energy in the large, but the compiler may not know the number of loop iterations and energy costs of data movement so automation of these decisions is not optimum. The same goes for may other compiler options [2] and garbage collection policies [3].
5) Crosstalk between the application of interest, other applications and peripherals cannot be deciphered and masks actual effects.

Modern debug and trace solutions like ARMs Coresight and UtraSoC's UltraDebug are an important step towards improving such visibility for processor cores and the system as a whole. They achieve this at the expense of silicon area. In the *Dark Silicon* era, where only a limited proportion of on-chip resources can be activated at any time, we argue that such non-computational uses for these abundant resources can significantly improve energy visibility and hence power efficiency.

Unfortunately, current methods for energy measurement and feedback to programmers are very crude and limited: power measurement APIs include only coarse-grained aspects such as battery voltage or total circuit current. Our *spEEDO* project proposes and investigates more precise, fine-grained instrumentation (and control) over power/energy at multiple levels of abstraction. The goal is to enable new energy optimization schemes, and increase the effectiveness of existing schemes by facilitating better coordination and collaboration between the programmer, the OS/runtime developer, and the hardware architecture. For software development, we provide infrastructure that allows programmers to identify energy-

intensive regions in their code. Regions can then be selected for optimization by various means, and the resultant effectiveness of the different optimizations attempted can be quantified.

This paper proposes:

1) extending the SoC components with (energy/power/-time) EPT registers that *record* loads incurred by their various host/parent components and system activity measurements provided by the on-chip debug support infrastructure;
2) correlating these measurements with *customer-numbers* that distinguish between the multiple application contexts that might be utilizing the component, so that the EPT for each component and each running application can be distinguished in hardware;
3) and extending hardware debug watchpoint mechanisms for finer-grain accounting within the work of a given **customer** or **flow of control**.

Ultimately these results are exposed at the higher levels of abstraction through a newly devised API for fine-grained energy measurement.

II. Related Work

Current methods of energy instrumentation and control: System operation can be monitored using software and or hardware instrumentation. Software instrumentation is where the program is statically or dynamically augmented in order to output information about the system. Although intrusive, software has the advantage of flexibility and established capabilities for various operating systems and debuggers. Examples include: LTTng [4] an established toolkit for Linux that uses static software instrumentation; and GDB tracepoints, an emerging dynamic instrumentation tool[1]. LTTng outputs its trace to memory or disk although it is also intended for streaming, whilst GDB uses its remote stub connection. Another example comes from ENTRA, where programs running on the XMOS core were profiled using custom extensions to the LLVM compiler chain [5]. Another popular tool is `dtrace` which is based on binary patching of previously inserted NOPs[2].

Hardware-based monitoring using on-chip debug support infrastructure has the advantage of being non-intrusive and is able to measure the activity for as many parts of the system as have been provided with suitable monitoring circuits and counters. In practice both software and hardware monitoring are needed to gain a comprehensive understanding of the system to determine its energy usage.

PC motherboards incorporate system monitor chips, such as the Winbond W83781D. These measure supply *voltage* as well as chip temperatures and fan speeds. Crucially they do not measure supply *current* and hence give no energy figures. Likewise, the standard BIOS APIs such as ACPI do not provide any energy measurement calls.

Until recently, the main means of determining energy use for a computer program involved time-consuming additional instrumentation, such as setting up a mains inlet current meter or resistive droppers in DC supplies. Likewise, laptop and Android developers could use a basic battery 'Gas Gauge' API such as the *Advanced configuration and Power Interface Specification* (ACPI) [6]. These techniques can provide some insight towards the gross total energy use for the system, but offer a measurement bandwidth of less than 10 Hz and cannot easily be used for investigating particular aspects of energy consumption.

However, most computer systems contain a good number of hardware event counters, either embodied in profiling hardware or in software device drivers. When hardware counters are provided, software tools like `oprofile` for Linux can show where time is being spent with minimal software overhead. The open-source **Gator** driver from ARM is an example where on-chip software reads hardware counters from both CPU and GPUs. It uses Ethernet in SoC devices to stream the data to an off-chip performance viewer called Streamline which can also accept board-level power measurements from an external hardware probe [3] Other debug tool providers such as Lauterbach offer probe-based power measurements which they correlate with fully decoded CPU instruction traces [7]. Holistically monitoring a heterogeneous SoC is also a major challenge, because each processor type typically has its own close-coupled debug support architecture [8].

For energy accounting, the obvious counters to use monitor the major architectural events, including retired instructions, branch mispredictions, cache misses and evictions at each level and DRAM row and column operations. But recently, Najem et al. explained how careful automated placement of event counters on apparently arbitrary nets of a SoC could collect sufficient information for an accurate power spline to be computed [9]. Counts must be combined in a polynomial with the instantaneous supply voltage to get energy figures.

A significant recent deployment is Intel's RAPL (Running Average Power Limit) announced in 2010. [10]. RAPL allows measurement of SoC power at a medium granularity of four domains: all cores, graphics, package, and DIMMs. Is neither fine-grained, nor application centric: i.e. it cannot provide figures on a per-application basis. The hardware API consists of several machine-specific registers (MSRs). These contain values computed by a microcontroller in the 'Sandy Bridge' subsystem that applies calibration weights to hardware event counters with the weights being determined or trimmed at system reset. Since family 15 (Bulldozer), AMD has provided similar total core power monitoring. AMD recommends that details are hidden from the O/S by the BIOS with the BIOS essentially refusing to honour a request to move to a DVFS levels that would exceed design parameters [11].

Measurement of energy use in large systems and datacentres is facilitated using energy logging frameworks such as the Energy-Aware COmputing Framework (EACOF) that provides real-time remote access to a centralised SQL database of energy and power events [12]. Such frameworks are handy for making dynamic datacentre management decisions at a macroscopic level: for instance, whether to power up another rack of server blades. Our current work can be a source of energy information for EACOF database and we have built shims above and below it. However, SQL transactions themselves use significant energy so this cannot serve for low-intrusion power debugging.

[1]GDB Tracepoints `https://sourceware.org/gdb/onlinedocs/gdb/Tracepoints.html`.

[2]Dtrace web site `http://dtrace.org+`

[3]ARM's Optimize `http://ds.arm.com/ds-5/optimize/`

Currently there is no established solution for energy monitoring other parts of the system such as its custom accelerators, interconnects and memory controller. The best system energy estimates are most easily obtained using power-annotated compiler chains. Running the object code on a virtual execution platforms gives additional insight provided the platform is calibrated. A virtual platform example is the PRAZOR simulator that is built in SystemC using TLM POWER3 library [13] and which is used for the practical experiments reported in this paper.

What is needed is a universal debug support platform that integrates with established proprietary debug frameworks and provides the hardware monitors needed for the rest of the system. By providing a hardware API, high-level software instrumentation from projects such as LTTng can also be combined with hardware instrumentation. The authors of this paper previously developed such a universal on-chip debug support platform and here enhance it for energy instrumentation.

III. The spEEDO API

The spEEDO API is designed to be implemented to one of several possible granularities on any particular platform or variant of that platform. For instance, a virtual platform (simulation model) might include richer support than the taped-out chip. Or better accuracy might be possible when an off-chip debugger is connected than is offered to on-chip applications. But Dark Silicon means we can include substantial monitoring infrastructure in real implementations.

The significant aspects of our complete spEEDO infrastructure are:

1) The system is composed of subsystems (IP blocks). An IP block might be a processor core, a cache, a coprocessor, a DRAM controller or a network interface, etc..
2) Energy information is recorded separately for each IP block, nominally within that block.
3) Moreover, the originator of the work that incurred the work is trackable so energy can be accounted on a **customer number** basis.
4) Energy information is carried through the system either by the debug infrastructure or else on the main busses by programmed I/O commanded by cores. Clearly the latter approach is more intrusive, but is appropriate in use cases where the O/S is dynamically load balancing on an energy basis.
5) The watchpoints and programmable F.S.M.s of the debug infrastructure are combined with the O/S knowledge of which jobs are active on which cores to facilitate detailed energy analysis during multi-processing and for areas of interest in software programs.

Fig. 1 shows various abstractions of the spEEDO API at four levels of abstraction. Starting at the bottom, interface **4** is a **Register API.** A typical fragment of the register API is shown in Fig. 2. This is accessed by a spEEDO device driver (or HAL component) running on a SoC core. The standard debug infrastructure will enable remote reads and writes to a core's I/O space, thereby providing a means of debug access to this view. The register API can also be mapped into debug address space of a core if the core has such a concept. The register API consists of some number of hardware contexts (minimum of 2) as well as a single read-only bank of registers that provide basic meta information such as the version of the API and the units used for time and energy.

Energy is dynamically consumed by a core as it reads and writes the register API. So, to avoid mis-read races, an atomic snapshot facility for the actual registers is provided. In the most simple form of the interface, there are just two contexts, one which changes all the time while the other is a hardware snapshot of the dynamic one triggered by a special write to a control register. In more-complex implementations, the host can alter which bank of registers is currently being incremented. However, in our multi-processing use case (described later) there may still be fewer hardware contexts supported than currently running processes on the local core, in which case the scheduler must context swap the hardware contexts when it makes a process switch. This is just an extension of the normal procedure that keeps track of how much CPU time each core has used. The number of contexts supported is reflected in the read-only portion.

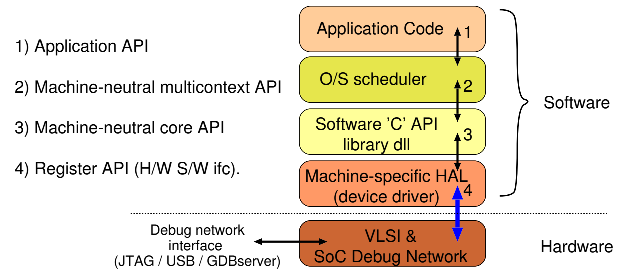

Fig. 1: The spEEDO API has various physical manifestations at different architectural boundaries.

Interface **3** is provided by a spEEDO **hardware abstraction layer** that abstracts the register API for high-level language use (typically C). This hides the nastiest machine-specific details and provides basic r/w operations on the abstracted register file. Simple use for a single process on a single core is as follows:

```
typedef struct // The record for constant data
{
  u32_t units;
  const char *spEEDO_get_reflection_uri;
} spEEDO_metainfo_t;

typedef struct // record for a snapshot of dynamic
    data.
{
  u32_t core_or_customer_no;
  u32_t local_energy;
  u32_t customer_energy;
  u32_t global_energy;
} spEEDO_local_record_t;

int spEEDO_get_meta(spEEDO_metainfo_t *&p);
int spEEDO_get_simple(spEEDO_local_record_t *&p,
        int cust=-1);
```

An important feature is the separation of local and global energy in the report. Both are running totals since system reset and the local is a subtotal figure that is included in the total. The distinction is that the local energy account records energy use from operations originating on the local CPU core. Ideally, the sum of local accounts across the system equals the global value.

```
//Constant registers:
#define SPEEDO_REG_MONICA        0   // Contains an identifying constant.
#define SPEEDO_REG_ABI           8   // Version number of the interface
#define SPEEDO_REG_ENERGY_UNITS  16  // This is the fraction of a Joule in the energy registers.
#define SPEEDO_REG_CMD_STATUS    40  // Command capabilities for resetting totals etc. & also the current H/W context.
#define SPEEDO_REG_TIME_UNITS    56  // Number of femtoseconds for each tick in the time register(s).
#define SPEEDO_REG_CORE_ID       64  // Core / Customer Number Identifier
#define SPEEDO_REG_CTX_CTRL      72  // Low 8 bits is no of h/w contexts (ro), bits 15-8 are current active context (r/w).
// The active context is the one being updated in h/w. The remaining contexts are passive.
#define SPEEDO_REFLECTION_URL0   1024 // First location of a canned URL giving further information on this ABI

#define SPEEDO_REG_CTX0_BASE     512         // active
#define SPEEDO_REG_CTX1_BASE     (512+256)   // shadow for easy read of 64-bit values over 32 bit bus.

// In this implementation there is only one context per CPU core
// but it is visible in active and shadow forms.  The active CTX is CTX0 and a snapshot of it is copied
// to CTX1, the shadow, as a side effect of writing any value to the CTX_CTRL register.

// Each hardware context contains the following time-varying registers (volatile):
#define SPEEDO_CTX_REG_LOCAL_ENERGY  8  // Running local energy in the units given
#define SPEEDO_CTX_REG_LOCAL_TIME    16 // Running local time (if implemented) for the context in the time units given
#define SPEEDO_CTX_REG_GLOBAL_ENERGY 24  // Running total energy in the units given - includes local energy
```

Fig. 2: Typical fragment defining the machine-specific registers (MSRs) for programmed I/O operation of the register API by host cores.

Our implementation also provides a few convenience functions at this level that return the total energy and average power consumption as double precision quantities in Joules and Watts. The user can subtract energies from successive checkpoints to account for specific intervals.

```
extern double spEEDO_local_energy_sofar();
extern double spEEDO_total_energy_sofar();
extern double average_power_sofar();
// ... and other obvious calls ...
```

Interface **2** provides **multiple contexts** in a machine independent way, even if the hardware only supports one or a few contexts. It also enables to read off the local energy figures for remote cores. For independent energy accounting, each process hosted by the O/S kernel requires an extended task control block. This contains not only the traditional register file image, priority and CPU time used accounts for a process, but also a running energy total. The O/S must save and restore the energy totals and also manage hardware customer tags (described later) over this interface.

Interface **1** provides a **virtual energy context** to a process that may be scheduled over multiple cores. Like the level **4** per-core register interface, the local and global energy are presented as separate running totals, only the local energy is for process and not a core.

IV. USE CASES

Case 1a: Programmer-directed Efficiency Optimizations: Intrusive Profiling: A programmer wishes to evaluate energy-efficiency benefits from offloading computation from the CPU to the DSP or GPU. He will recode key regions of code using portable concurrent programming models such as OpenCL. The old and the new code are made available by conditional compilation. An energy checkpoint is taken before and after the region of interest. We created an application shim library for the EACOF framework [12] to map the energy checkpoints using the spEEDO API so the example in Fig. 3 can also be coded using EACOF primitives in the same style.

```
...
spEEDO record_t before, after;
spEEDO_chkpt(before);
if (USE_GPU) use_gpu(); else old_version();
spEEDO_chkpt(after);
spEEDO_record_t delta = after-before;
cout << "Energy␣used␣" << delta.toString()
     << "\n";   ...
```

Fig. 3: Use Case 1a - Simple, but intrusive, bracketing around a test.

This is an intrusive measurement because the energy checkpoints include new code within the application. The same measurement can be made via the debugger interface using breakpoints or unintrusive watchpoints. Connecting to our virtual platform with the GDB debugger we can set break or watchpoints on the two labels we have inserted in the binary object file of the program (Fig. 4). Using symbolic labels is clearly easier than finding hex addresses by hand and the `volatile` keyword stops the compiler reordering around the region of interest.

```
...
asm volatile ("start_point1:"::"memory");
if (USE_GPU) use_gpu(); else old_version();
asm volatile ("stop_point1:"::"memory");
...
```

Fig. 4: Use Case 1b - Inserting code labels for debugger watchpoints.

```
$ gdb
(gdb) target remote :9600
(gdb) break start_point1
(gdb) break stop_point1
(gdb) run; cont; cont
...
```

```
(gdb) eptDiff
GLOBAL_ENERGY = 32.02mJ, TIME=5.03ms, ...
```

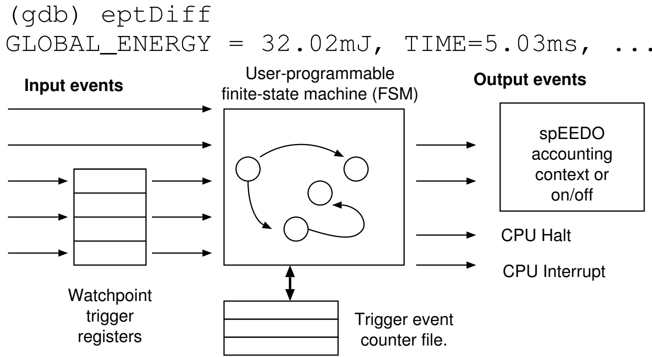

Fig. 5: Generalised watchpoint and sequencer structure.

Various useful debug extensions are implemented by installing Python script files in GDB. These know the addresses of the relevant spEEDO registers in the debug spaces of the various cores. They also install hooks that read these registers and record state inside the debugger. Various commands have been implemented, such as `eptDiff` that gives the energy used between the last two breakpoints.

Case 1b: Non-Intrusive Profiling: Despite minimal modification to the user's program, stopping the machine at breakpoints for tens of milliseconds while the debugger executes a number of read commands is intrusive to any sort of concurrent system. Also, static power continues to accumulate while cores are stopped. The **Watchpoint API** fixes this. Given that most non-trivial controllers and processors contain hardware watchpoint registers it is sensible to exploit the watchpoint infrastructure for selective energy accounting. In the past, the output from the watchpoint mechanism simply made the core or all cores enter sleep mode so that debug cycles could be run. This has become too invasive in modern multicore SoCs and more flexible output routing is desired, such as the ability to freeze instruction trace buffers as well as general counter operations and issue remote core interrupts. A more modern and general scheme is illustrated in Fig. 5 that consists of a user-programmable FSM. Inputs come from watchpoints and other architectural events. A counter file is included so the nth event or other complex trigger sequences can be matched. In order to trace energy only between two program counter values, that respectively pre- and post-dominate a code section of interest, the two addresses are placed in the watchpoint registers (by remote access debug cycles or local host operations). The FSM is then programmed to enable and disable local energy accounting outside that region.

Case 2: Multitasking. Most SoCs today use a multiprocessing scheduler. We need to solve two problems: 1. making sensible scheduling decisions and 2. cleanly reporting energy use to a process that roams over many cores and peripheral devices without *crosstalk* from other applications. A multithreaded O/S where the threads are dynamically mapped to cores can use an extended scheduler to keep energy accounts for each thread group or process.

Since IBM's VM-360, the concept of a *process tag* has existed in many hardware architectures. Their original purpose was for hardware protection and to avoid a TLB flush on a context swaps. The idea is that extra process identifier bits are associated with each virtual address. In today's 64 bit architectures this can be reasonably implemented as otherwise unused high-order address bits, but generally they are held in a CPU control register and concatenated in hardware with every effective address generated by the application program. Modern on-chip busses also have the ability to carry user tags and other *user sideband* signals alongside the main address and data information. These tags may be used to match up out-of-order bus transaction results. As a specific example, the AXI interface from ARM allows `awuserm[AWUSER_WIDTH-1:0]` to be declared. Additionally, load-linked instructions commonly used to achieve atomic transactions in a NoC environment require each originator to supply an identifier that is stored at the target in readiness for the store-conditional operation. Together, these examples show that the idea of carrying a customer number, in-band, as part every bus transaction is not overly far-fetched. Indeed, for virtual platform use, the TLM POWER3 library extends the transactional general payload with a customer number that serves as a process tag.

Given this infrastructure, the scheduling operating involves mapping process identifiers to a physical core and process tag. At context swap time, the root translation pointer is adjusted with the new context. This commonly contains the process tag in its lower bits. Also, the system timer is noted to implement the CPU time account for that task. Our observation is: it is very little further overhead to also save and load the running energy totals from the spEEDO register API at a context swap. In this way, per process energy accounting is provided.

V. IMPLEMENTATION COMPLEXITY

We do not have room for an extensive discussion of implementations in this paper. But we need to justify that our API is feasible.

Dark Silicon presents a landscape where a subsystem might have its own boot-time embedded microcontroller with its own RAM and ROM (e.g. Thacker's DDR2 controller for the BEE3, or the microcontroller that implements the on/off switch on many contemporary laptops) so quantity of logic is not a major issue *per se*. Considerable quantities of ROM are also very cheap to implement and are commonly copied to L2 cache during processor boot for a multitude of start-of-day procedures, such as DRAM leveling and secure key validation. Adding further boot-time complexity is not a problem.

In the most simple implementation of our API, each originator has some local accounting mechanism, such as several counters for retired instructions and load/store front-side cycles. If each core additionally has an L1 cache miss counter we can get some measure of how much shared resource energy to allocate to that customer core. Accuracy will be further improved if we can get L2 misses into L3 and L3 misses into DRAM accounted on a per customer basis. However, the *correct answer* is not precisely defined: who should be charged for capacity and sharing evictions? It is complicated to charge the evictor for energy burnt by a customer who must reload a cache line he had already loaded. Regardless or that, we also assume the chip as a whole has PSU energy instrumentation (or at least the current DVFS information can be read). With calibration information, this basis is sufficient to provide the simple local and global energy counters of the basic spEEDO API.

A microcontroller can potentially make online computations using the above information served over the on-chip debug bus/network. A kilobyte, say, of microcontroller ROM should provide a quality energy figure, computed from time to time, as per RAPL. However, we argue that sub-millisecond energy reports are likely to be useful, and if several of these are requested at different IP blocks at the same time, a solo microcontroller might be overloaded. Clients may have to stall waiting for the computation (that they could indeed compute more quickly themselves). Therefore, a hardware implementation is probably preferable overall and we are implementing this at the moment. It is the 'energy digestor' in Figure 6. But this hardware still uses coefficients measured by software at boot-time and recomputed at supply voltage change time.

Turning to the complexity overhead for shared caches and peripheral devices, the most precise information can be collected where process tags are conveyed with the bus traffic and shared resources have banked event/energy counters indexed by process tag. However, outside of fully-instrumented virtual platforms, the hardware overhead of having a large number of banked counters may be unpalatable. For instance, typical cores from Intel and AMD at the moment have 250+ events that can be monitored in hardware, but provide fewer than ten hardware counters for the complete chip. The Xilinx Zynq ARM 7 cores have six counters that can allocated each to one of fifty or so event sources. A routing API is used by programs like `oprofile` to wire the counters to events of current interest. A similar approach can be taken by the spEEDO implementation. A minimal implementation within a cache or IP block is to deploy a counter that counts for just one process tag serves to effectively filter out traffic of current interest. If that one tag is programmable so much the better, otherwise the O/S must dynamically reallocate the tags when the process of current interest rotates.

VI. Virtual Implementation

We are starting to constructed silicon embodying the spEEDO API and concepts (§VII), but so far we have mostly just extended a high-performance virtual platform so that all aspects can be simulated and some power-aware applications can be run. A novel aspect of our platform is that it was already fully annotated with energy logging for each major operation, such as an instruction fetch, mis-predicts, cache operations at multiple levels and a DRAM energy model using the University of Maryland DRAM simulator [14]. Therefore we did not have to implement power supply monitors and event counters explicitly, we were able to simply connect the spEEDO API to the energy instrumentation of the simulator.

The virtual platform is implemented in SystemC using the TLM 2.0 transactional modeling style. It supports various CPU architectures, including ARM, x86_64, MIPS64 and OpenRISC. It can be set to be binary compatible with the Xilinx Zynq series and then boots the same Linux binary and SD card image. Standard benchmarks, written in C/C++, can also be compiled with a minimal implementation of `libc` and pthreads to run essentially bare metal on the platform. Power annotation is via the TLM POWER3 library calls [13]. This library was augmented with the customer number concept. Each TLM packet contains a customer number in its payload alongside the normal data, address and control fields. The payload was already augmented beyond the default TLM 2.0 generic payload with fields to estimate bit switching activity so that transaction wiring energy could be logged on a hamming distance and bus length basis.

The virtual platform also embodies a relatively simple debug network. A TCP connection to a GDBserver port on the running simulator runs the RSP protocol of `gdb`. Although this protocol does not readily support multicore systems and non-uniform address maps, it does support the concept of switching between active threads. We abused this facility, so that the number of cores was augmented by the number of processes over all the cores by the debugger stub and hence the user could connect to an actual thread or to an actual core using suitable `thread` commands. The debugging traffic is routed over the virtual platform through instantiated debug components that are no different in their modeling style from the main bus components: i.e. they are SystemC class instances with TLM sockets statically wired to form the on-chip debug network.

The virtual platform is called PRAZOR and is available with its spEEDO implementation from `http://www.bitbucket.org/prazorvhls`.

VII. Physical Implementations

A version of the spEEDO API will be implemented in RISC-V multicore test chip currently being designed in our labs. Figure 6 illustrates the general structure in simplified form. The 'energy digestor' is implemented in hardware, reading performance management (PMU) registers and power supply instrumentation to populate the spEEDO register contents.

In the meantime, while the test chip is in development, we are implementing a mock up in the Zynq 7010 FPGA. The hardware boards are augmented with high-fidelity power sensing hardware that separately measures the core supply to the Zynq chip and the rest of the card (which essentially consists of the Ethernet MAC and the DRAM devices). Unfortunately we are unable to separately measure the consumption of the individual ARM cores but we see this as a common physical limitation in future chips, even where spEEDO has been designed in, owing to supply sharing across cores. But, an RTL implementation of the digestor can read the PMU registers for each core, as well as for some shared resources, like the L2-cache and DRAM. Using this information, online energy estimates are generated, as outlined above. The spEEDO API is then provided as AXI slave registers that can be read by the software on each core: notably the extended Linux scheduler, or over the JTAG interface. The scheduler context swaps the energy readings using the same code as when it runs on the virtual platform. We are also inputting to the FPGA the digital control signals from the various switching power supplies in the system, since these can be combined with a single ADC reading of the unregulated input supply voltage to get individual rail energy use with reasonable accuracy (e.g. $\pm$ 15 percent).

VIII. Demo: DOTNET Virtual Machine Hotspot Detection

As an end-to-end example of energy optimisation across the stack, we took applications coded in C# targeting the DOTNET virtual machine. We used C# but Java/Dalvik could be used in the same way and would be a more relevant example

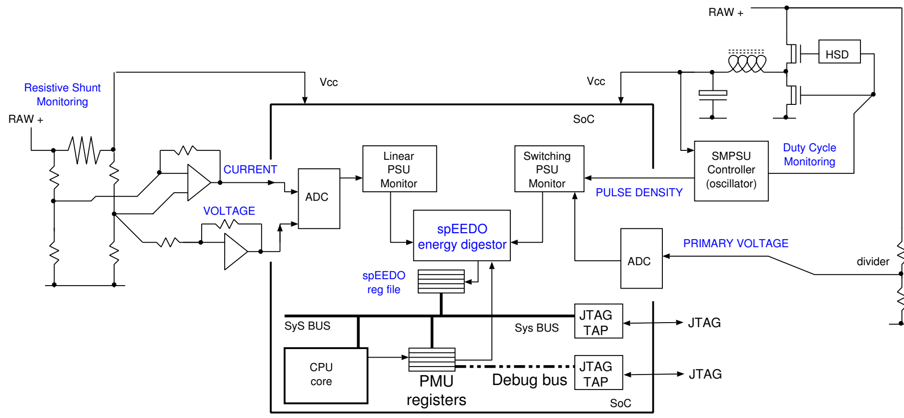

Fig. 6: RTL/Hardware Energy Digestor and Instrumentation of Power Supplies.

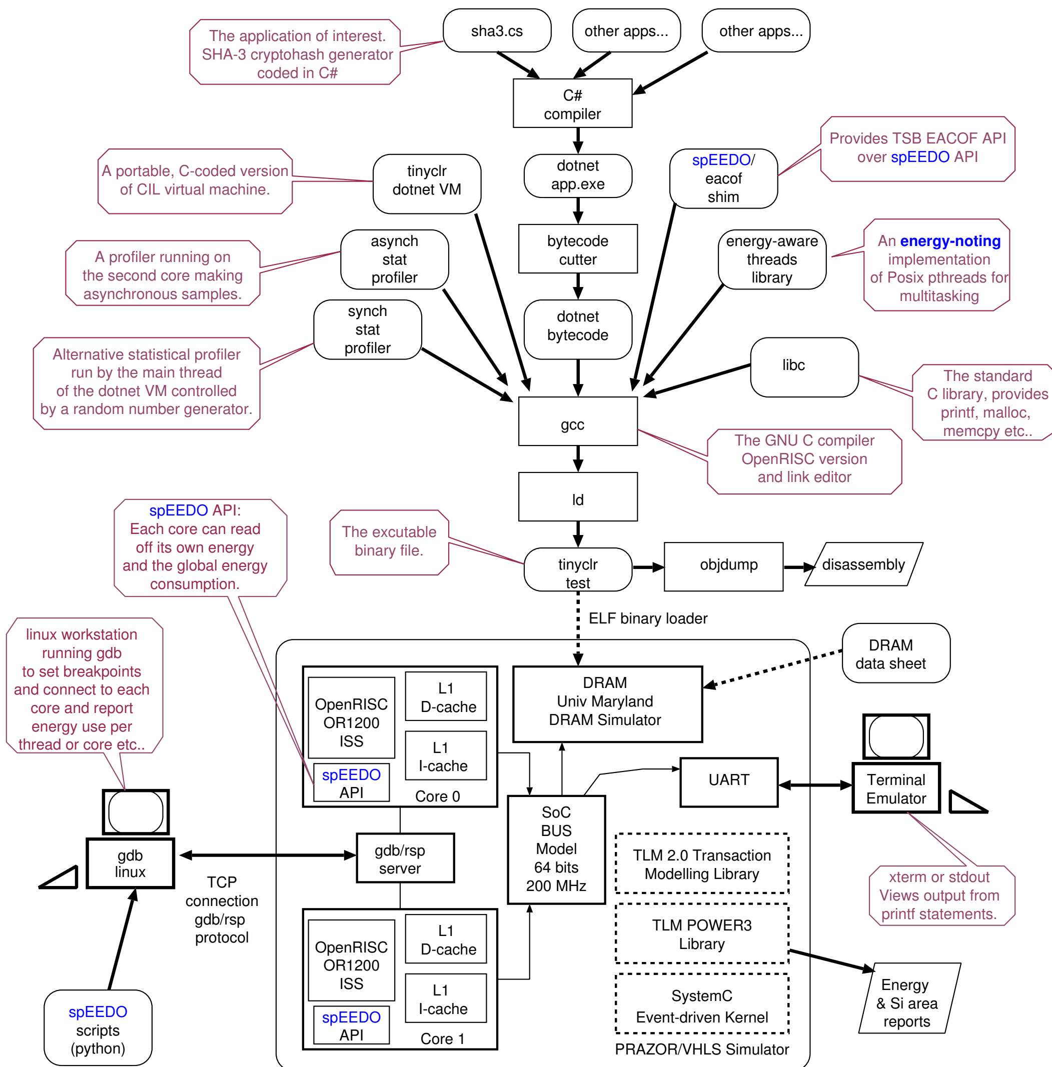

Fig. 7: Demonstration Example: Power Profile for DotNet and Energy Hotspot Detector.

for mobile phone app energy profiling. The purpose of the demo is to show that the same application-level and debugger APIs can be used for both the real silicon and the virtual platform and to discuss the engineering decisions that might be made based on this information and the differences that might arise between the two platforms. Using a small subset

of C# we were able to get away with a locally-coded version of the VM that is itself energy aware in two ways:

1) it allows the C# code to invoke the spEEDO API for high-level application purposes, and
2) it embodies an energy profiler that reports not only the time in each method but the energy used in each method.

The VM uses one operating system thread per virtual thread. An asynchronous energy monitor was implemented using a further processes-level thread that will typically be running on an otherwise unused core, assuming more cores are present than threads in the C# application. The monitor wakes up every 10 milliseconds and inspects the state of the worker threads, logging their energy and execution statistics to a memory data structure. The asynchronous energy monitor gives more accurate readings for short-running method calls given that the overall test runs long enough to get a sufficient number of samples. This is because the sub-millisecond invocations suffer systematic rounding errors owing to the correlation arising when samples are taken on the same core/thread as the workload: with a digestor update rate of 1kHz many short-lived ones will always indicate zero energy use on the real hardware owing to aliasing. The virtual platform has much-finer energy logging in its POWER3 library implementation, so does not suffer from that.

Because the VM threads are mapped to O/S threads, and because the O/S is saving energy information at context swaps, a unified energy accounting system exists from application space right down to cache line fills from DRAM. If the VM had used a user-space threads package then that too would have had to be augmented with energy context accounting.

When GDB is connected to the virtual platform, the energy registers of the hardware cores are directly inspectable and the O/S energy contexts are inspectable. They are printed in human-readable form by the scripts running inside the debugger. When the Zynq mock up is properly working, we should see exactly the same output from the physical platform.

For software developers, the energy information can guide which functions are hot spots that might best be JIT compiled to native code. However, so far, we have not seen any notable cases where a decision based on energy use would lead to a different decision compared with a decision based only on cycle count. Nonetheless, this information can be used to guide efficiency decisions relating to dynamic micro-architecture adaptation, such as having segments table entries that indicate the best number of ways in a set associative cache to enable for that region of memory [15]. For hardware developers, the energy information can be used as a guide for moving code to FPGA or custom hardware coprocessors — the conservation cores approach [16].

IX. Conclusion and Further Work

An extended draft of this paper is being shared with research teams at a number of SoC companies and we await feedback. Using our virtual platform we have conducted experiments testing compiler optimisation levels, making intelligent scheduling of work over cores, and examining crosstalk isolation between concurrent applications in a busy environment. As mentioned, our FPGA implementation of the digestor is ongoing and an ASIC test chip is being designed.

Acknowledgment

The authors would like to thank the UK TSB (now Innovate UK) for their support and funding.

References

[1] H. Esmaeilzadeh, E. Blem, R. St Amant, K. Sankaralingam, and D. Burger, "Dark silicon and the end of multicore scaling," in *Computer Architecture (ISCA), 2011 38th Annual International Symposium on*. IEEE, 2011, pp. 365–376.

[2] J. Pallister, S. J. Hollis, and J. Bennett, "Identifying compiler options to minimise energy consumption for embedded platforms," *CoRR*, vol. abs/1303.6485, 2013. [Online]. Available: http://arxiv.org/abs/1303.6485

[3] A. Diwan, H. Lee, D. Grunwald, and K. Farkas, "Energy consumption and garbage collection in low-powered computing," Tech. Rep., 2002.

[4] D. Toupin, "Using tracing to diagnose or monitor systems," *Software, IEEE*, vol. 28, no. 1, pp. 87–91, Jan 2011.

[5] Roskilde University, University of Bristol, IMDEA Software Instituted & XMOS Limited, "A general framework for resource consumption analysis and verification," European Union $7th$ Framework Programme FP7-ICT-2011-8, Tech. Rep., November 2013.

[6] F. P. Miller, A. F. Vandome, and J. McBrewster, *Advanced Configuration and Power Interface: Open Standard, Operating System, Power Management, Cross- Platform, Intel Corporation, Microsoft, Toshiba, ... Sleep Mode, Hibernate (OS Feature), Synonym*. Alpha Press, 2009.

[7] L. G. Elmar Stahleder. (2014) Optimization of embedded softwares energy consumption. [Online]. Available: http://www.lauterbach.com/doc/optimization_of_embedded_software.pdf

[8] A. B. T. Hopkins and K. D. McDonald-Maier, "Debug support for complex systems on-chip: a review," *Computers and Digital Techniques, IEE Proceedings* -, vol. 153, no. 4, pp. 197–207, July 2006.

[9] M. Najem, P. Benoit, F. Bruguier, G. Sassatelli, and L. Torres, "Method for dynamic power monitoring on FPGAs," in *Proceedings of 24th International Conference on Field Programmable Logic and Applications, FPL'14.*, 2014.

[10] H. David, E. Gorbatov, U. R. Hanebutte, R. Khanna, and C. Le, "Rapl: Memory power estimation and capping," in *Proceedings of the 16th ACM/IEEE International Symposium on Low Power Electronics and Design*, ser. ISLPED '10. New York, NY, USA: ACM, 2010, pp. 189–194. [Online]. Available: http://doi.acm.org/10.1145/1840845.1840883

[11] (2013, January) BIOS and kernel developers guide for AMD family 15h models 00h-0fh processors. [Online]. Available: http://support.amd.com/TechDocs/42301_15h_Mod_00h-0Fh_BKDG.pdf

[12] H. Field, G. Anderson, and K. Eder, "EACOF: A framework for providing energy transparency to enable energy-aware software development," in *Proceedings of the 29th Annual ACM Symposium on Applied Computing*, ser. SAC '14. New York, NY, USA: ACM, 2014, pp. 1194–1199. [Online]. Available: http://doi.acm.org/10.1145/2554850.2554920

[13] D. Greaves and M. Yasin, "TLM POWER3: Power estimation methodology for systemc tlm 2.0," in *Models, Methods, and Tools for Complex Chip Design*, ser. Lecture Notes in Electrical Engineering, J. Haase, Ed. Springer International Publishing, 2014, vol. 265, pp. 53–68. [Online]. Available: {\tthttp://dx.doi.org/10.1007/978-3-319-01418-0_4}

[14] P. Rosenfeld, E. Cooper-Balis, and B. Jacob, "Dramsim2: A cycle accurate memory system simulator," *IEEE Comput. Archit. Lett.*, vol. 10, no. 1, pp. 16–19, Jan. 2011. [Online]. Available: http://dx.doi.org/10.1109/L-CA.2011.4

[15] C. Dubach, T. M. Jones, and E. V. Bonilla, "Dynamic microarchitectural adaptation using machine learning," *ACM Trans. Archit. Code Optim.*, vol. 10, no. 4, pp. 31:1–31:28, Dec. 2013. [Online]. Available: http://doi.acm.org/10.1145/2541228.2541238

[16] G. Venkatesh, J. Sampson, N. Goulding, S. Garcia, V. Bryksin, J. Lugo-Martinez, S. Swanson, and M. B. Taylor, "Conservation cores: Reducing the energy of mature computations," *SIGARCH Comput. Archit. News*, vol. 38, no. 1, pp. 205–218, Mar. 2010. [Online]. Available: http://doi.acm.org/10.1145/1735970.1736044

This paper was presented at FDL-2015 Barcelona, September 2015.