西 北 工 业 大 学

博 士 学 位 论 文

（学位研究生）

题目：暗硅时代 CoDA 架构
可扩展性及能效问题研究

作　　者：郑乔石

学科专业：计算机科学与技术

指导教师：高德远 教授

2015 年 1 月

Exploring Energy and Scalability in Coprocessor-Dominated Architectures for Dark Silicon Regime

By
Zheng Qiaoshi

Under the Supervision of Professor
Gao Deyuan

A Dissertation Submitted to
Northwestern Polytechnical University

In partial fulfillment of the requirement
For the degree of
Doctor of Computer Science and Technology

Xi'an, P. R. China
January 2015

摘 要

硅工艺朝着物理极限的不断迈进，导致了由摩尔定律和登纳德定律组成的集成电路传统缩放模型失效。在芯片功耗墙的限制下，人们发现在后登纳德定律时代，芯片设计中存在使用墙问题以及由此所观察到的暗硅现象。更进一步地，随着工艺的持续进步，暗硅现象会不可避免地急剧恶化，使得芯片设计进入暗硅时代。

在暗硅时代，芯片上可以在极限时钟频率下翻转的晶体管的比例急剧下降，这使芯片上出现大量无法有效利用的晶体管。这些不断增加的无法使用的晶体管，导致在设计芯片时功耗和能耗与芯片的面积相比更为重要。这种设计思路的转变导致了利用暗硅来换取高能量效率的新型体系结构不断涌现，大量集成异构专用协处理器就是其中之一。

单个专用协处理器与通用处理器相比可以提高 10 倍以上的能量效率，使得集成少量专用协处理器的系统能量效率大大提高。但常见的系统具有大量不同的应用负载，为了提高这样系统的能量效率，架构师需要集成大量的异构专用协处理器并调度软件到专用协处理器上执行。这使得最终系统架构成为 CoDA（Coprocessor-Dominated Architecture）。

本文紧紧围绕作者作为 GreenDroid 和暗硅团队成员在加州大学圣迭戈分校工作期间，所进行的论证 CoDA 架构设计合理性、可扩展性、能量效率、发现解决未来 CoDA 架构实现所遇到的潜在问题展开，进行了以下几个方面的创新性工作：

（1）研究了 CoDA 对应用的适用性，并以此说明 CoDA 适合暗硅时代。本文分析了安卓移动软件栈，发现大部分应用是基于共享原生库和虚拟机的，硬件化这部分软件就可以使得应用的大部分运行在专用协处理器上。之后重点分析了安卓浏览器，并使用硅构造专用协处理器实现了这个浏览器。实验结果表明在 22nm 工艺下 7mm^2 的硅面积用于构造专用处理器就可以覆盖浏览器 90%的运行。使用可接受的硅面积就可以覆盖应用执行，证明了 CoDA 架构适合暗硅时代。

（2）针对快速探索 CoDA 设计空间的需求，提出了 CoDA 架构分析模型，并对本文提出的多维度可扩展 CoDA 架构进行建模。该架构可以由不同数量的瓦片组成，每一个瓦片可以包含不同数量的函数粒度专用协处理器，并且每一个专用协处理器都可以是异构的。分析模型用来评估每一种特定 CoDA 架构的能量、面积和性能；模型参数既包含了高层次的体系结构参数，也包含低层次的电路实现参数。

（3）探索了 CoDA 架构在不同 Cache 配置、瓦片大小、粗粒度能耗管理策略以及晶体管实现等参数下的能量效率问题。在最优化的参数条件下，与通用架构相比小规模 CoDA 设计可以带来 5.3 倍的能量效率优化和 5 倍的能量延时积（energy-delay product, EDP）优化；而对于支持上百个应用的大规模 CoDA 设计，可以带来 3.7 倍的能量效率优化和 3.5 倍的 EDP 优化。这说明为大规模应用而设计的大规模 CoDA 扩展是有效的。

此外，本文发现 CoDA 设计即使采用了激进的能耗管理策略，漏电功耗所占总功耗的比例仍然随 CoDA 规模增大而增大。

（4）探索了并发执行对 CoDA 能量效率的影响。积极的影响是这些同时运行的程序或线程可以分摊漏电功耗等固定的开销，这样可以提高系统的能量效率。消极的影响是，当驱动 CoDA 生成的目标应用集合和实际运行的应用集合不匹配时，会造成大量程序竞争某些专用协处理器，系统的平均能量效率将大大降低。本文提出 CoDA 架构集成覆盖多个函数功能的融合 QsCore 来减少竞争冲突。实验表明使用融合 QsCore 的方式，仅仅增加 41%的面积就可以提供 2 倍数量的专用协处理器，并使得非均匀分布负载的能量效率提高 11.1%~22.1%。

（5）针对使用当前工艺实现的 FPGA 模拟下一代工艺实现的 CoDA 芯片时，单个 FPGA 芯片资源不足的问题，提出了跨多芯片可扩展的 2D-mesh 片上网络。该网络由跨芯片的环形网络连接，并为跨芯片的每一个 2D-mesh 物理通道分别提供跨芯片的流控机制。跨芯片的环形网络提供了 ASIC 芯片到 FPGA 以及 FPGA 之间两种可选连接方案。通过使用该设计方案，本文使用两块 Virtex 6 FPGA 芯片首次实现了 CoDA 架构原型系统。

关键词：暗硅，CoDA，大规模异构，协处理器，能量效率，可扩展性

Abstract

As silicon technology approaching its physical limitation, the traditional scaling theory guided by Moore's Law and Dennard's Law is about to fail. Under the limited power budget, we find the Utilization Wall and Dark Silicon phenomenon exist in current chip designs in the Post-Dennard scaling era. Furthermore, the dark silicon phenomenon will worsen precipitously with each process generation, making the chip design go into the dark silicon regime.

In the dark silicon regime, the percentage of a chip that can switch at full frequency is dropping precipitously, which leaves more and more on chip transistors couldn't utilize. So silicon area becomes less expensive relative to power and energy consumption. This shift calls for new architectural techniques that trade dark silicon area for energy efficiency. One such technique is the use of heterogeneous specialized coprocessors.

Because the specialized coprocessors could save more than 10x energy than general-purpose processors, so employing coprocessors could improve the energy efficiency of the system for single application. However, most of the common systems have a great number of diverse workloads, in order to improve the energy efficiency of such system, architects need to employ hundreds or even thousands of specialized coprocessors and schedule software to run on these coprocessors. As the number of coprocessors scales up, these designs will transform from coprocessor-enable systems to *Coprocessor-Dominated Architectures (CoDAs)*.

As a member of UCSD GreenDroid group and Dark Silicon Center, the author writes this paper at UCSD. This paper focus on the theory, scalability, energy efficiency, and some potential issues about the CoDA. The innovations include the followings:

(1) This paper makes the feasibility study of CoDA, and demonstrates CoDA is suitable for the Dark Silicon regime. This paper analyzes the Android mobile software stack, and finds most applications running on native libraries and virtual machine. If we build coprocessors for these shared codes, most of the software will run on coprocessors. Then, the paper analyzes web browser, and uses silicon to build it. The results show that it only need 7mm^2 chip area to cover 90% web browser dynamic execution on specialized coprocessors under 22nm process. Its only take an acceptable piece of silicon area could cover most of the application execution, which indicate that CoDA is suitable for the Dark Silicon regime.

(2) In order to explore CoDA's design space under acceptable speed, the paper proposes a CoDA analysis model. The CoDA architecture in analysis model could employ a

multi-dimension scalable structure. In this paper, CoDA could compose by different number of tiles, each tile can contain different number of coprocessors and each coprocessor could be heterogeneous. The analysis model could evaluate the energy, performance and area of each specific CoDA design, and the design parameters includes both high level architecture configurations and low level implementation configurations.

(3) Exploring the CoDA energy efficiency under different Cache configurations, tile sizes, coarse-grained energy management strategies and the transistor libraries. Under the optimal configuration, CoDA design approach that can deliver 5.3× improvements in energy efficiency and 5.0× improvements in energy-delay product for small workloads could continue to yield improvements of 3.7× in energy and 3.5× in energy-delay for designs covering over 100 applications. A scalable CoDA design can continue to deliver superior efficiency even for large workloads, which means CoDA could scale. In addition, the paper finds even with aggressive power management, leakage is still a sizable fraction of CoDA energy that grows with coprocessor count.

(4) Exploring the influence of concurrent execution on CoDA's energy efficiency. The effects are divided into positive and negative sides. On the positive side, running multiple threads on a CoDA increases overall energy efficiency because it amortizes fixed energy costs, including those due to leakage, across the work from multiple threads. On the negative side, when the target workloads drive CoDA generation mismatched the real workloads, concurrent threads raise the possibility of competition for c-cores, which could reduce the average energy efficiency of the system. The paper proposes to integrate the merged QsCore into CoDA to reduce the competition conflicts. The results show that using QsCore to provide twice number of C-cores for each type, only cost 41% additional area, but it could improve the energy efficiency from 11.1% to 22.1% in the non-uniform case.

(5) Because single FPGA chip implemented on current technology process does not have enough recourse to emulate the CoDA chip target on next generation process. The paper proposes an inter-chip scalable 2D-mesh network, which connected by cross chip ring network. The inter-chip design also provides flow control for each physical channel of the 2D-mesh. The ring network design provides two types of connectors for crossing the chip. One is ASIC to FPGA (MURN IO) connector; the other one is FPGA to FPGA (FMC) connector. By using the inter-chip 2D-mesh network, the paper uses two Virtex 6 FPGA boards to set up the CoDA prototype system at the first time.

Key words: Dark Silicon, Coprocessor-Dominated Architecture (CoDA), Massively Heterogeneous, Coprocessor, Energy Efficiency, Scalability

目 录

图索引

表索引

1 绪论

1.1 研究背景

1.1.1 使用墙问题和暗硅时代

深亚微米工艺出现之前的 30 多年，VLSI 的发展几乎严格遵循着经典的摩尔定律[1]和登纳德定律[2]，这也是芯片设计师的美好时代。在这 30 年中，仅仅凭借 CMOS 工艺的进步，就可以使得各种运算设备的性能越来越强、体积越来越小甚至功耗也可以逐渐降低。工艺每进步一代，就会使得晶体管的面积减少一倍，晶体管的翻转速度也会按照特征尺寸的收缩比例而提升，并且由于供电电压的下降单个晶体管翻转所需的功耗也会随之成指数下降。这些变化使得在芯片总功耗近似不变的情况下，芯片上就可以集成 2 倍的晶体管。

在这 30 年中，凭借这些越来越多的晶体管，设计师可以在处理器中放置越来越多的执行单元、功能部件和采用更加复杂的机制来不断提升处理器核的性能。比如使用复杂的超标量流水线、乱序执行、线程级前瞻执行、复杂的分支预测等等。

近年来，当工艺进入深亚微米之后情况发生了改变。虽然芯片上集成晶体管的密度还在不断成指数增长，但是单个晶体管翻转所需的功耗不再随特征尺寸的降低成指数下降。这导致了在固定的芯片功耗预算下，可以在最高时钟频率下翻转的晶体管数量占比随工艺的进步成比例下降。这就产生了使用墙问题[3-6]，并由此观察到暗硅的现象[7-13]。接下来本文首先给出使用墙的含义，之后再从理论分析、工业界产品观察来阐述这个问题的合理性和真实存在性。

学术界所说的使用墙问题（Utilization Wall）是指由于功耗的限制，每一次工艺的更新，将导致芯片上可以以最高速度翻转的晶体管的比例成指数降低。使用墙问题的产生是由于摩尔定律和登纳德定律共同组成的经典缩放模型中的登纳德定律失效造成的。

DRAM 的发明人登纳德（Robert H. Dennard）[2]在 1974 年指出每一次工艺的更新将带来 S^2(本节假设晶体管缩放因子为 S，约为 1.4)倍的晶体管，这些晶体管的运行频率可以提高 S 倍，所以相同芯片面积下潜在的运算能力将提高 S^3（2.8）倍。与此同时，单个晶体管的电容将降低为原来的 1/S，内核电压也降低为原来的 1/S。由此计算出的芯片上所有晶体管的总功耗不变。总体来看登纳德定律阐述了两个问题：1）处理器的计算性能提升并不是仅仅受益于数量越来越多的晶体管，也受益于晶体管越来越快的翻转速度。2）登纳德定律表明了使用新工艺后，同样面积的芯片功耗几乎不变。

公式（1-1）给出了单个晶体管的功耗计算公式。晶体管的功耗由三部分组成，等式右侧第一项为动态功耗，第二项是漏流功耗，第三项是短路功耗。由于在 130nm 工艺之前，晶体管的功耗主要取决于动态功耗，所以使用新工艺实现的单个晶体管功耗降低为原来的 $1/S^2$。但是芯片上可以集成原来 S^2 倍的晶体管，所以芯片的总功耗可以维持不

变。这就是登纳德定律的第二个结论。具体计算过程见表 1-1 中登纳德模型列。

$$p = \frac{1}{2}\alpha C V_{dd}^{\ 2} f + I_{leakage} V_{dd} + I_{sc} V_{dd} \tag{1-1}$$

当工艺演进到 90nm 之后，晶体管的漏流功耗逐渐占据了主导地位。晶体管的漏电流主要由三部分组成：亚阈传导漏流、结漏流和栅极漏流。在低电源电压和低阈值电压的芯片上，亚阈传导漏流是晶体管漏电流的主要部分，而且会随着阈值电压的不断降低而按指数趋势增加[14]。

$$f_{\max} \propto (V_{dd} - V_t)^2 / V_{dd} \tag{1-2}$$

公式（1-2）是表征晶体管翻转速度的关系式，可见晶体管的翻转速度与供电电压和阈值电压的差值成正比。因此为了维持晶体管的翻转速度，V_{dd} 的降低必然引发 V_t 的降低。

$$I_{subthreshold} \propto \exp(-V_{th} / T) \tag{1-3}$$

公式（1-3）表征了亚阈传导漏流的关系式，可见亚阈传导漏流和阈值电压的变化趋势成反方向的指数关系。也就是说当阈值电压降低的时候，亚阈传导漏流将成指数增长。

综合公式（1-2）和（1-3），为了使亚阈传导漏流维持在可接受的范围以便获得可以承受的漏流功耗，就不能按照工艺收缩的比例降低阈值电压甚至需要维持阈值电压不变。在这种情况下，为了维持晶体管翻转速度以及获得足够的噪声容限，芯片的供电电压也不能按照传统模型的收缩比例降低。供电电压的缓慢降低甚至维持不变，将导致芯片整体的功耗快速上升，这使得登纳德定律失效。

表 1-1 传统缩放参数与后登纳德定律缩放参数对比[8]

CMOS 晶体管属性	登纳德定律时期	后登纳德定律时期
功耗预算	1	1
芯片面积	1	1
特征尺寸缩小比例（W, L）	1/S	1/S
晶体管数量变化	S^2	S^2
晶体管频率变化	S	S
晶体管电容变化	1/S	1/S
芯片内核电压变化	1/S	1
单个晶体管功耗=αFCV^2	$1/S^2$	1
芯片总功耗=晶体管数 x 单个晶体管功耗	1	S^2
使用率=1/芯片总功耗	1	$1/S^2$

表 1-1 的第三列给出了在深亚微米工艺下登纳德定律失效后，各个参数收缩的状况。这里假设芯片的供电电压维持不变，那么最终芯片上所有晶体管完全翻转的总功耗将变为 S^2。这样在芯片功耗预算维持恒定的假设前提下，芯片上晶体管的使用率将降低为原来的 $1/S^2$。这是使用墙问题出现的理论基础，也由此开启了后登纳德定律时代。

在后登纳德定律时代，摩尔定律仍然发挥作用。这也就是说工艺每进步一代，芯片上的计算资源所提供的潜在能力依旧提高 2.8 倍（晶体管数量增多并且速度变快，S^2xS）。

但是在使用墙的限制下，仅仅只有 1.4 倍的潜在性能可以得到发挥。其他提供额外性能的晶体管或被关断电源或被彻底关断时钟，由此导致了架构师所看到的暗硅现象。如果假设在当前工艺下，芯片上所有的晶体管同时翻转所需的功耗为芯片的功耗总预算并维持不变，那么根据后登纳德定律时期的收缩公式，8 年后的芯片上（假设经过 4 代工艺）将有 93.75%的晶体管处于暗硅状态。图 1-1 是 ITRS[15]在 2012 版报告[1]中回顾的其在 2001 年预测的暗硅占芯片面积比例图。这和上述理论分析的结果具有类似地趋势。此外，由于漏电功耗是 CMOS 器件的基本属性，如果没有器件的突破性创新，暗硅现象还要继续恶化下去，从而不可避免的使芯片设计进入暗硅时代[2]。

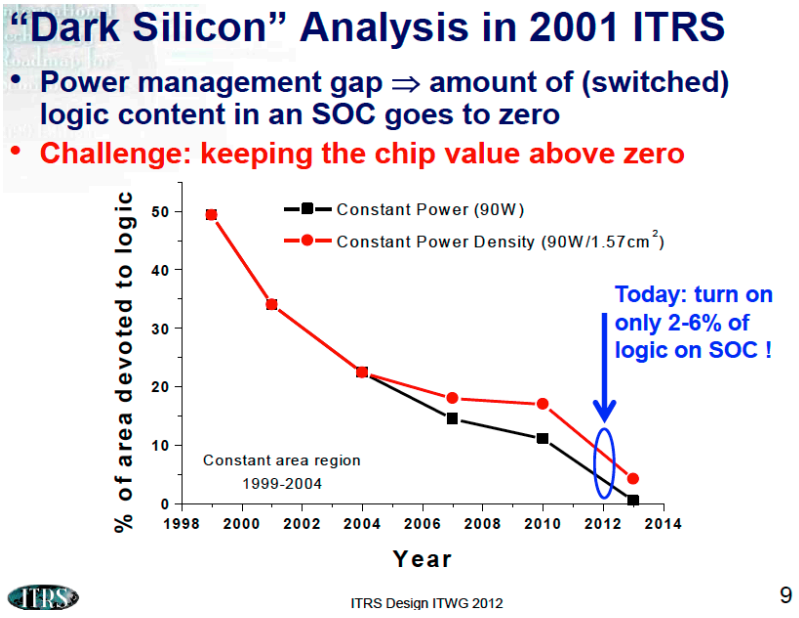


图 1-1 ITRS 2012 年关于暗硅占芯片比例预测回顾[15]

体系结构的设计需要跟随芯片所能提供资源的变化而变化。在登纳德定律时期，架构师使用摩尔定律带来的额外晶体管来追逐最高的性能；今天在后登纳德定律时期，架构师将使用这些额外的晶体管来追逐更高的能量效率（energy effciency），以此来适应暗硅时代芯片上的资源。

进入后登纳德定律时期的最直接的表现就是工业界的设计在 2005 年开始转向多核。多核的设计思路见图 1-2。65nm 到 32nm 的转变经过了 2 代工艺，晶体管的缩放参数 S=65/32 约等于 2。那么理论上，芯片上可以集成 4 倍数量的处理器核，并且需要 4 倍的功耗来维持这些晶体管逻辑正常工作。然而芯片的功耗不能无限上涨，在维持芯片功

[1] ITRS 2012 年的预测报告是在 2013 年发表，由于 2013 年的预测报告在本论文撰写时还没完全整理发表（2015 年 1 月重新查看过），所以 2012 的报告为目前可见的最新预测报告。
http://www.itrs.net/Links/2012Winter/1205%20Presentation/DesignSD_12052012.pdf

[2] 学术界对于“暗硅时代”的英文提法有多种，包括“Dark Silicon Era”、“Dark Silicon Regime”和“Dark Silicon Age”。本文作者较为倾向于“Dark Silicon Regime”，其想表达的意思是暗硅不可避免，需要体系结构去适应。但是由于没有发现较为恰当合适的汉语词汇对应，所以本文的中文还是说“暗硅时代”。此外，本文所说的解决暗硅问题也是适应暗硅时代的意思。

耗大体不变的约束条件下[1]，从 65nm 工艺到 32nm 工艺架构师可以将处理器的核数增加到 2 倍，但是要维持处理器频率不变（图中灰色部分），这也是当前大多数工业界的选择。另一种方法是维持核数不变，将处理器的主频提高两倍。其他还有一些基于两者之间的权衡,例如核数增加一半，频率也增加一些。但是不管采用哪种方式，芯片上的大部分资源都是无法有效利用的，这些资源就成为了暗硅（图中黑色部分）。

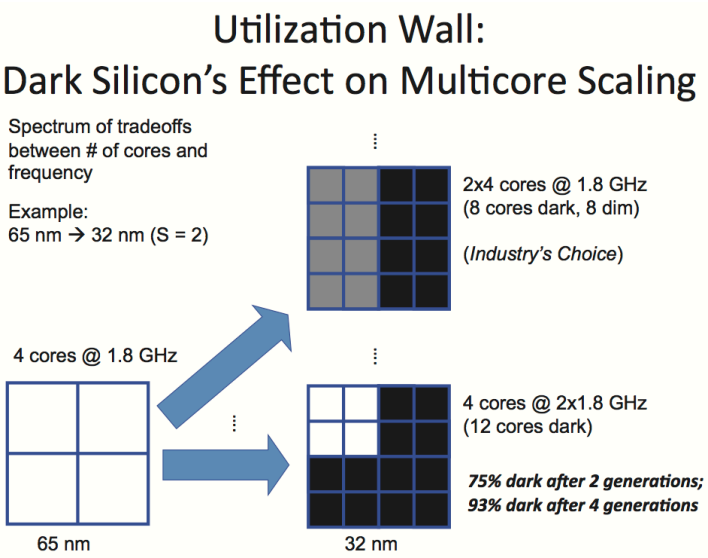


图 1-2 通用多核扩展所带来的大量暗硅[10]

对最新的高端主流商用处理器产品的观察也可以证明这一点。在桌面端，对 Intel 4 代 I7 处理器 Nehalem、Sandy Bridge、Ivy Bridge 和 Haswell 的分析可以看出，I7 架构的发展基本符合使用墙的理论。即最高端的芯片维持主频缓慢增长的同时，慢慢增加处理器核的数量，也就是图 1-2 中两者之间的一种中间设计方案。在服务器端，观察苹果公司最新的 Mac Pro 服务器[16]搭配的 Intel 志强处理器可以发现核数越多的高端芯片，主频越低。例如 6 核的志强处理器主频为 3.5GHz，而 12 核的志强处理器主频仅为 2.7GHz[2]。在移动端这种趋势更为明显，例如华为海思的移动应用处理器麒麟 920 芯片采用 big.LITTLE 架构[17],这种架构可以根据应用的需求打开性能较强的 4 核 Cortex-A15 处理器，也可以打开高能量效率的 4 核 Cortex-A7 处理器，当然也可以在低频率下同时打开 8 个核。图 1-3 是商用处理器主频和时间的关系（来自 ISSCC 2014 年报告[18]），可以很明显的看出 2005 年工艺进入深亚微米之后，处理器的主频几乎不再增长。

无论采用图 1-2 中哪种解决方案，或者他们之间的任何一种折中方案，都无法缓解芯片上将出现越来越多暗硅的趋势。所以多核架构设计更像是一种处理器架构发展的中间状态，而不是解决使用墙的最终方案。目前已经有一些研究论证了多核结构将无法在

[1] 芯片的功耗预算不仅仅取决于芯片本身的散热、稳定性等因素，更取决于所在的系统。例如移动平台的系统总功耗要求 4W 左右，无线基站平台要求芯片最大功耗是 50W（华为海思）。

[2] 核数和主频的数值增减倍数并不一定严格成比例。这是因为芯片上除了处理器核之外，还有共享的三级 Cache，以及多种外设控制器，互联等等部件。

未来继续扩展[7-11]。那么为了继续跟随工艺发展的脚步，解决未来 12nm、8nm 以后处理器架构的问题，需要对处理器结构进行新一轮的创新，并且这一轮创新需要解决的主要问题不再是如何利用更多的晶体管来提高处理器的性能，而是如何有效利用这些晶体管来提高处理器的能量效率，使得设计也可以有效利用增加的晶体管进行扩展。

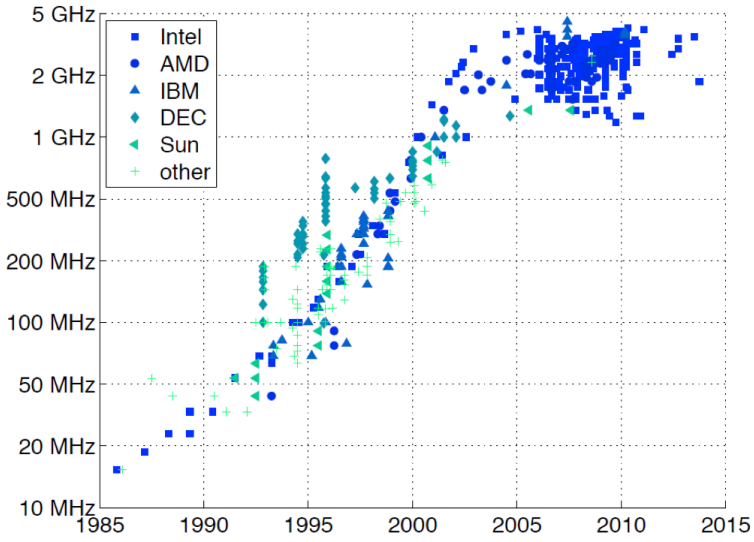


图 1-3 商用处理器时钟频率随时间的变化

暗硅并不是指空白的、无用的或者没有使用的硅；暗硅仅仅是没有一直使用的硅或者没有在极限频率下工作的硅[8]。在深亚微米工艺之前，芯片中也存在着暗硅（或者叫暗逻辑）。例如 Cache 就属于暗逻辑，因为 Cache 中的晶体管平均翻转频率远小于 1%；定点程序执行时浮点部件也成了暗硅。

本文认为如果底层器件没有突破性的创新，暗硅现象就不可避免并会越来越严重。适应暗硅时代的架构研究实际上就是指找到合适的系统架构使用更多的晶体管在满足能耗约束的前提下尽可能提高性能或者在满足性能需求的前提下提高能量效率[1]。因此本文所说的“解决暗硅问题”是指找到合适的架构适应暗硅时代芯片资源特点。

1.1.2 适应暗硅的架构研究

使用墙和暗硅的研究工作并非一帆风顺，经历了最开始的备受质疑、之后逐渐被大家接受以至今天成为研究热点。本小节将描述使用墙问题、暗硅研究的发展过程以及现状。

这里首先来探讨一下功耗墙和使用墙、暗硅研究的区别。尽管功耗问题由来已久，人们也相应地提出了功耗墙来说明功耗问题的严重性。但是功耗墙与使用墙、暗硅研究侧重点不同，功耗墙的侧重点在于芯片的功耗有一个无法逾越的上限，这个上限可能是由于芯片散热、稳定性、设备供电或者系统架构所决定的。使用墙和暗硅研究的侧重点在于芯片在功耗上限的限制下，未来芯片上的晶体管不可能同时全速翻转。这导致在芯

[1] 在暗硅时代直接面对的问题应该是解决能耗效率。

片工作的任意时刻，芯片上都出现大量的关断电源、关断时钟或者降低翻转频率的晶体管。使用墙和暗硅研究表明如果体系结构没有创新性的突破，那么架构师将无法利用新工艺所提供的大量晶体管资源，同时芯片的性能提升也将十分有限[1]。功耗墙的提出促使大量低功耗设计研究的出现，而暗硅的提出将导致系统架构发生突破性的改变，暗硅研究重点关注在未来 14nm、8nm 工艺直到达到物理极限时芯片的架构组织问题。

使用墙问题在 2006 年由加州大学圣迭戈分校（UCSD）的 Michael Taylor 和 Steven Swanson 教授总结发现，并于 2010 年在 ASPLOS 会议上由 UCSD 的 Michael Taylor 教授和 Steven Swanson 教授团队正式提出而进入人们的视线[5]。在 2010 年的 HotChips 会议上 UCSD 将使用墙问题进一步形象地抽象为暗硅（Dark Silicon）[6][2]。之后在 2011 年的 ISCA 会议上，华盛顿大学的 Hadi 分析了暗硅时代[11]，并对当前使用的主流架构进行了建模，之后在一定程度上分析预测了多核扩展时代将要终结。至此使用墙和暗硅现象才逐渐被人们所接受。在 2012 年 ISCA 会议举办时，UCSD 的 Michael Taylor 教授、Steven Swanson 教授和 Jack Sampson(现为宾州州立大学助理教授) 一起组织承办了首届 Dark Silicon 会议（DaSi）。与此同时，在 DAC 2012 年的会议上，UCSD 的 Michael Taylor 教授总结了所有一切潜在的适应暗硅时代的架构方法[10]，并分析了每一种方法可能遇到的挑战。在这次会议上 Michael 还提出了 CoDA（Coprocessor-Dominated Architecture）的概念，这种 CoDA 架构就是本文研究的主要对象。2013 年 UCSD 的 Michael Taylor 教授和 Steven Swanson 教授作为特邀编辑，编辑出版了一期 IEEE Micro 专刊，该专刊重点讨论了面向暗硅时代的研究。之后使用墙和暗硅研究才逐渐成为热点。从发表的文献数量来看，使用墙和暗硅的研究从 2010 年到 2013 年每年发表的重要文献从几篇逐渐增加到十几篇，进入 2014 年发表的重要文献上升到几十篇

总体来说，目前适应暗硅时代的架构研究有以下四个方向[3]：1）通过在芯片中集成专用逻辑；2）使用淡硅的方式(Dim Silicon)，扩展核数但是降低部分主频或电压；3）使用新器件替代 MOSFET；4）近似计算。

专用化是目前学术界较为关注的方案。UCSD 的 Michael Taylor 教授[4]、Steven Swanson 教授[5]和 Jack Sampson 领导的团队尝试了这个方向（本文作者为该团队成员）。

UCSD 的团队首先在 2010 年的 ASPLOS 会议上发表了文献[5]，在这篇论文中首次提出了使用墙问题，这篇论文主要从理论分析的角度定义并引出了使用墙，之后从实验

[1] Intel 最新的芯片已经出现了 Die 在逐渐变小的趋势，说明其无法在功耗限制下利用摩尔定律提供的全部晶体管来提供更好的性能。

[2] Dark Silicon 的名词并不是由 UCSD 首次提出，ARM 公司的 CTO Mike Muller 在 ARM 技术会议上提到了这个名词,但没有公开发表的文献。参见 http://www.eetimes.com/document.asp?doc_id=1172049，关于暗硅这个词汇更为详尽的历史请参见 Prof. Michael Taylor 的主页。

[3] 本文对暗硅研究的分类与作者外导师 Prof. Michael Taylor 稍微不同，去掉了 Die 收缩的方向，添加了近似计算。

[4] 博士生阶段为麻省理工 Raw 处理器主要架构师。作为最早的同构众核架构处理器，Raw 提出了瓦片结构的概念。

[5] 博士生阶段为华盛顿大学 WaveScalar 处理器主要架构师

和对商用处理器的观察等角度论证使用墙问题真实存在。论文提出了 Conservation-core (本文简称 C-core)的最初架构，这种专用协处理器从设计一开始就追求更高的能量效率。之后的论文[13, 19]对 C-core 的架构进行了升级和改进，加入了一些既能提高性能又能提高能量效率的方法，例如选择性去流水化等等。文献[20]介绍了新版本的 C-core 自动生成工具。

之后，作为 2010 年 HotChips 会议 SoC 部分唯一一篇由高校完成的论文，UCSD 的论文[6]将使用墙问题抽象成暗硅（Dark Silicon），并提出了集成 C-core 为安卓系统搭建 GreenDroid 的初步想法。该项目的后续工作发表于论文[4]，以及本文作者发表的论文[3]，这些论文更为完整的介绍了 GreenDroid 的想法，并且增添了更加详实的实验数据。此外，这些论文还初步介绍了 GreenDroid 的验证系统和流片工作。目前最为详细的 GreenDroid 硬件实现工作见本文第五章。

2012 年 DAC 和 DaSi 上发表的论文[10]预测了未来处理器的设计将更加注重能量效率而不是性能的提升。这篇文献也分析了适应暗硅时代的四个潜在研究方向，并且简单综述了这四个方向中当前的研究现状以及难点。更为完善的总结和论述参见 Michael 的论文[7-9]。后续的关于暗硅的研究通常都是沿着这四个方向进行，并且该论文中也提出了 CoDA 架构的概念。

微软研究院的 Doug Burger 教授[1]、威斯康星麦迪逊的 Karu 教授[2]以及华盛顿大学 Hadi Esmaeilzadeh（现为佐治亚理工助理教授）领导的团队同样也探索了专用化方向。

该团队的理论研究情况发表于 2011 年的 ISCA 会议。该论文[11]论证了在暗硅条件下，目前的主流多核架构将因为无法提高性能而即将终结。该论文对各种不同的主流多核架构进行了较为详细的建模分析，认为未来制约性能提升的主要原因就是暗硅。该文献预测了类似 CPU 的架构，集成超过 35 个核之后继续集成更多处理器核系统性能提升将微乎其微。 此外，该文献还发现，这种多核可扩展的架构仅仅对具有高度并行性的负载（99%以上为并行部分）来说性能才可以持续提升，而对于普通的应用 8nm 工艺实现的处理器仅仅可以比现在提升 3.7 倍的性能。

文献[21, 22]是威斯康星麦迪逊 Karu 教授提出的解决方案。DySER 在通用处理器的流水线中添加了新的可配置的功能单元，并通过将经常执行的控制复杂代码和并行代码映射到配置阵列上来提高系统的性能和能量效率。

在 2012 年 1st DaSi 会议上哥伦比亚大学的 Lisa Wu 和 Martha A.Kim 发表的论文[23]针对不同的测试程序集进行了细致的程序分析，得出了以下三点结论：1）通过分析 SPEC2006 测试集，证明了 C-core 和 DySER 这样的加速器对于非结构化的 C 代码是有效的；2）对 Java 测试程序集的分析认为对于这种面向对象编程语言可能面向类而不是

[1] 博士生阶段在威斯康星麦迪逊参与了 SimpleScalar 仿真器的开发，在德州大学奥斯丁分校做教授期间领导了 TRIPS 处理器的研发。
[2] 博士生阶段在德州大学奥斯丁分校参与 TRIPS 处理器的研究工作。

单独函数的加速器可能更为有效；3）认为用专用处理器来填充暗硅，系统将要集成几十或者几百个加速器。该论文不但证明了 C-core 的有效性，而且也从侧面论证了 CoDA 研究的必要性。这也是本论文研究合理性和迫切性的基础。该论文提出的解决方案属于专用逻辑的解决方案。

弗吉尼亚大学发表的文献[24]探索了暗硅时代的专用化研究方向。该论文建模并权衡了使用加速器和可重构逻辑的优缺点。论文表明对于具有理想并行化的多个不同负载，可重构逻辑比加速器具有更好的能量效率。这是因为可重构逻辑在多个不同应用间具有可重用性，并且可以提供更好的加速度。

哥伦比亚大学和 Cadence 公司一起研究了暗硅时代加速器的存储器被其他通用架构处理器或者专用加速器共享使用的问题[25]。该研究也属于专用化研究方向。

加州大学洛杉矶分校的学者在文献[26]中，针对暗硅时代大量使用加速器的架构提出了优化的加速器与共享存储器互连方案。该研究同样探索了专用化研究方向。

淡硅研究方向通常降低供电电压并降低处理器的主频。宾夕法尼亚大学和密歇根大学发表的文献[27-30]，探索了淡硅方案。该研究小组提出了一种较为独特的解决方案，他们探索了负责芯片散热的各种材料，并使得芯片在某一段时间功耗可以超过芯片的功耗预算。这段时间是以芯片达到的正常操作温度上限为终止。采用这种方法可以使得芯片在处理延迟敏感的任务时加快速度，这样尽管在短时间内会消耗大量的能量，但是可以使芯片尽快完成任务，并更快进入休眠的状态，所以在整体上会节约大量的能量消耗。

密歇根大学的论文[31]探索了暗硅时代的淡硅方向。该论文提出的解决方案是在架构中集成更多的处理器核，这些处理器核如果使用普通电压是无法完全供电的；所以相应地，解决方案降低了工作电压到接近阈值电压的水平（near threshold voltage），使得芯片的总功耗控制在预算内。之后，为了维持系统的性能，就需要使程序尽可能地并行化来使用这些更多的处理器核。该论文讨论了这种方法的优缺点。

犹他州立大学的暗硅时代研究团队在文献[32, 33]中提出了多核拓扑上同构功耗性能异构的体系结构（Topologically Homogeneous Power-Performance Heterogeneous, THPH）。这种多核架构是由一些体系结构角度相同的处理器核组成，但是每一个核针对不同的电压-频率（voltage-frequency）组合进行优化。这样当任务需要较高的性能时，可以迁移到高电压高频率的处理器核上执行；与之相反，如果任务不需要较高性能，可以迁移到较低电压低频率的处理器核上执行，这样就可以节省大量的能量。这个研究是解决 Dark Silicon 的淡硅方案。

新器件的研究是最无法预测的。宾州州立大学的学者尝试了研究新器件。文献[34]尝试将隧道场效应管（tunnel field-effect transistors, TFET）引入处理器逻辑中。与 MOSFET 相比 TFET 的阈值电压更低，并且可以节省大量的能量。但是在高电压状态 TFET 的速度比 MOSFET 慢。这篇论文提出了在多核设计中同时包含 MOSFET 工艺实

现的处理器核和 TFET 工艺实现的处理器核。

近似计算也较为新颖，该研究方向的主要想法是在仅仅需要近似解的时候仅仅计算出近似解而不是精确解来降低能量消耗。文献[35-38]是微软研究院和华盛顿大学提出的解决暗硅的近似计算方案。近似计算（approximate computing）也在 2012 年被 ITRS 列入了低功耗设计技术路线图，这种技术是说在有些计算仅仅需要近似解或者满足基本需求的情况下，可以将精度或者正确性降低，这样就可以节省大量的能耗。适合近似计算的应用包括视觉[39]以及机器学习[37]等等。在系统架构上近似计算需要编程模型、编译器、处理器架构和电路等多个层次的支持。

其他一些研究还包括：

宾州州立大学和北京大学一起展开了在暗硅时代的片上网络研究[40]。新南威尔士大学（澳大利亚）和卡尔斯鲁厄理工学院（德国）的学者也共同研究了暗硅时代的高能量效率片上网络[41]。

卡尔斯鲁厄理工学院（德国）、滑铁卢大学（加拿大）和卡耐基梅隆大学的学者在文献[42]中提出了解决暗硅时代芯片热量、可靠性以及对工艺变化容忍的一系列方案。

图尔库大学（芬兰）的学者在文献[43]中研究了在暗硅时代基于片上网络的众核架构处理器的在线测试调度算法。

新加坡国立大学研究了在暗硅时代big.LITTLE 非对称多核架构的功耗管理问题[44]。

国内体系结构研究一直领先的国防科大也在张春元教授的带领下展开了暗硅时代微处理器设计的研究。

此外，未来架构使用的适应暗硅的技术可能会组合使用这 4 个方向的研究成果。例如将同时包含专用逻辑、不同功耗-性能优化的通用处理器核以及不同器件实现的逻辑等等。

人类大脑作为暗逻辑比例最高的计算架构之一，提供了一种计算机体系结构颠覆性创新的新方向[8]。今天的计算机仍然无法完成很多人类大脑可以处理的任务，尤其是视觉相关的工作。大脑具有 800 亿个神经元和 100 万亿个突触，并在低于 100 毫瓦的功耗下工作。大脑证明了高度并行、可靠和存在大量暗逻辑的结构是存在的，并且大脑还呈现了低操作频率(Dim)，专用化和新器件的特点，这些都与现在暗硅的研究类似。与处理器相比神经元的操作频率特别低，最快的情况下也仅仅可以达到 1KHz。尽管神经元和晶体管具有本质的属性区别，用硅模拟的神经元在做计算时会带来过大的“解析”开销，但是大脑可以作为暗硅时代架构的参考。大脑提供了一种重新设计系统来满足暗硅时代所需的极低操作频率和极低工作电压的深入、长远的研究方向。

1.2 CoDA 研究工作

1.2.1 CoDA 架构

CoDA(Coprocessor-Dominated Architecture)是最近由 UCSD GreenDroid 团队的 Prof.

Michael Taylor 提出的适应暗硅时代芯片资源特点的新型架构。该架构的基本想法是将软件中的常用函数、共享库、程序基本块等程序段硬件化成专用协处理器，并集成到芯片架构中。极限情况是为所有应用负载的每一个程序段设计一个专用协处理器。这样每当程序执行到这些硬件化的函数和基本块时，程序就跳转到相应的专用协处理器上执行。为了将功耗控制在预算内并最大限度地提高能量效率，理想情况下 CoDA 仅仅为正在计算的处理器核供电，其他等待的逻辑全部断电。这样随着程序的执行，程序使用的一系列专用协处理器核被依次点亮再关闭（闪烁）。

CoDA 架构与人类大脑也有相似之处。首先是专用化，人脑中由神经元组成的不同区域有不同的功能。有的具有不同的认知功能、有的连接不同的感觉器官，并允许重构，随着时间的推移突触连接还可以面向计算进行定制。CoDA 中每一个专用协处理器都是一个晶体管集合并可以完成特定的功能。

其次是操作频率低，人脑中的神经元最大的“工作频率”为每秒钟 1000 次。在 CoDA 架构中操作粒度从指令变为专用协处理器，因此会大大降低计算逻辑的操作频率。在 CoDA 中某一个特定的专用协处理器仅仅在应用执行到对应函数时才工作。

第三是只有少部分逻辑“供电”，大脑中的神经元在同一时刻仅仅只有少部分工作而其他大部分神经元都是“暗逻辑”。在 CoDA 中由于操作粒度是专用协处理器，同一时刻也只有几个专用协处理器工作，其他大部分的专用协处理器都处于关断的状态。

目前本团队实际研究的 CoDA 架构大体按照上面的想法设计。架构中不但有通用处理器，还有大量面向应用负载的专用协处理器，并且架构中专用协处理器的数量远远超过通用处理器的数量。在实际的 CoDA 架构中，程序在通用处理器和专用处理器之间跳转执行，跳转过程基于当前任务运行在哪种处理器上可以获得更高的能量效率。同时当前没有使用的器件进入深度的低功耗模式。在 22nm 或者更先进的工艺下，芯片的暗硅面积将提供足够的晶体管资源来构建 CoDA，这样的 CoDA 架构可以包含成百上千的专用协处理器。

大量集成专用协处理器使得 CoDA 设计即使面对大规模负载也可以保证程序的大部分运行在专用协处理器上，并通过不断集成更多专用协处理器的方式提高专用协处理器上的代码执行覆盖率。专用协处理器的代码覆盖率越高，负载在 CoDA 架构上运行的全局能量效率就越高。

CoDA 架构的提出基于以下一些事实。首先，随着暗硅时代的到来，芯片上出现了越来越多的无法同时供电的晶体管，这就像人脑中存在大量的神经元。这些晶体管使 CoDA 架构有足够的资源来专用化所有需要执行的程序段，并通过这种专用化全部功能的方式使系统通用化。这与人脑类似，整体上看人脑可以处理所有信息（通用），但是每一种信息都是特定区域来处理的（专用）。

其次，异构和专用化具有高能量效率可以有效的适应暗硅时代。CoDA 方案就是利

用这些暗硅来实现一系列的专用协处理器，这些协处理器可以比通用处理器速度更快[45]，也可以比通用处理器的能量效率更高（10-1000 倍）[5]。最近的研究已经提出使用这些暗硅来构建专用协处理器，每一个专用协处理器都比通用处理器降低 10 倍以上的能耗[6, 12]。此外专用协处理器对性能和功耗的优化和权衡空间比通用处理器更大。

第三，随着计算机编程技术的发展，应用软件的编写往往基于一系列的共享库函数，系统软件的编写也需要调用一系列的系统调用函数。这就使得芯片架构师可以面向这些共享库函数和系统函数设计函数粒度的专用协处理器，并在软件调用这些共享代码时跳转到专用协处理器上执行。通过这样的方式，就可以用适度的硬件代价来快速提高专用协处理器对应用负载运行时的动态覆盖，进而提高系统的能量效率。

专用协处理器的研究由来已久，几乎是伴随着通用处理器的发展而发展。早期的专用协处理器都是作为加速器而出现的，例如浮点协处理器；之后就出现了处理并行指令或者数据的加速器；近年来，由于芯片设计遇到越来越严重的暗硅现象，架构师又大规模使用专用协处理器来适应暗硅时代。

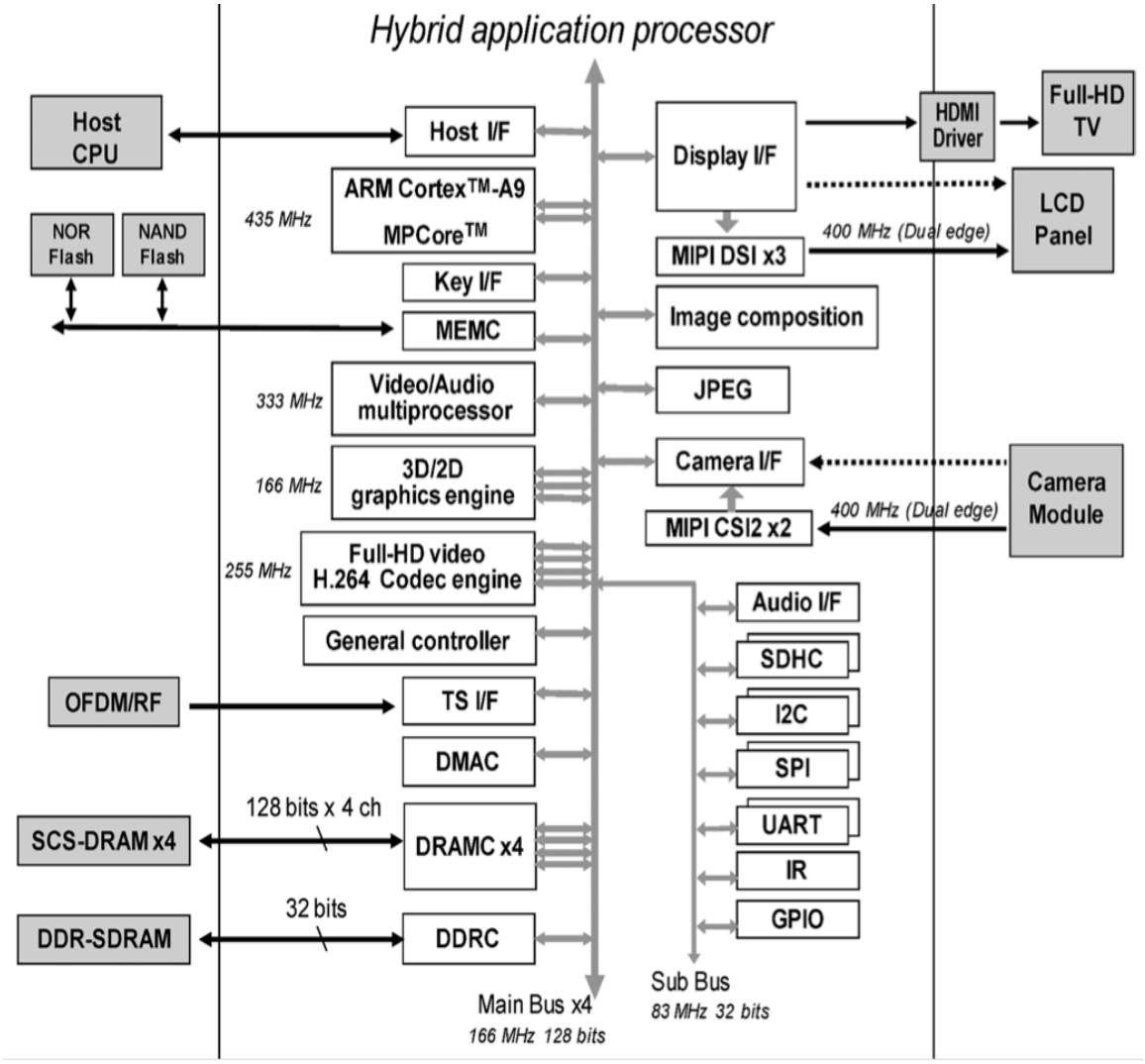


图 1-4 东芝 40nm 低功耗应用处理器架构

图 1-4 是东芝在 2011 年研发的移动应用处理器[46]，它集成了大量的专用协处理器。这些专用协处理器都是为某些特定应用而设计的，当运行这些特定应用时与通用处理器相比不但性能上可以大幅提高，而且能量效率也可以大幅提高。

虽然理论上 CoDA 架构可以有效利用暗硅，但是大规模向 CoDA 架构中集成专用协处理器还有以下几个潜在问题：

1）软件与硬件接口的划分。软硬件的接口实际就是软件工程师与硬件设计师之间的一种约定，只要遵守这个约定软件工程师编写的软件就可以正确地在硬件设计师设计的电路上正确运行。明确的是在通用处理器中，软件和硬件的接口就是指令集。但是当系统中出现越来越多的不同厂商设计的专用协处理器后，统一专用协处理器的软硬件接口将变得尤为重要。如果接口划分不明确就会导致两方面的问题：首先，为专用协处理器设计的编程语言可能无法在相同功能的专用协处理器之间兼容，例如 CUDA 语言无法兼容 AMD 和 Nvidia 的 GPU；其次，软件工程师编写或者移植程序到专用协处理器上将变的较为复杂，例如在 CELL 处理器上编程。

2）开发专用协处理器是非常复杂和繁琐的工作，需要大量工程师的人力投入以及大量资金支持。与采用复制的方式开发通用多核、众核处理器不同，通常芯片上集成的每一个专用协处理器都可以是异构的、具有不同的功能。这就需要工程师单独设计每一个专用协处理器。

3）大部分常见的加速器都是针对某类特定地并行程序而设计的。与论证程序加速的 Amdahl 定律[47]类似，如果仅仅并行程序可以获得较高的能量效率，而串行程序还是运行在通用处理器上，整个系统的能量效率提升将非常有限。

作者所在的 UCSD GreenDroid 团队在文献[5, 13]中提出的 Conservation Cores（C-core）自动专用协处理器生成技术基本上可以解决程序专用化时出现的上述三方面问题，这个技术也是 CoDA 架构提出的工程基础。

首先，专用协处理器的自动生成直接解决了第二个问题。虽然自动生成的专用协处理器在性能或者功耗方面比工程师手工优化的专用协处理器要差，但是与通用处理器相比仍然可以获得十几倍的能量效率优化。此外，工程师可以不断优化自动生成工具链来生成越来越好的专用协处理器，这将拉近自动生成电路与手工优化电路之间的性能以及功耗差距。

其次，C-core 兼容了软件进行程序调用的标准二进制接口 ABI，这使得通用处理器上运行的程序可以像调用函数一样调用 C-core，并使程序的执行由通用处理器跳转到专用协处理器。使用函数粒度的专用协处理器 C-core 并不需要对程序源代码进行修改，对程序员来说是透明的。这样不同厂商设计的专用协处理器就有了可以遵守的软硬件约定并方便了软件的移植。

第三，自动生成的 C-core 是面向串行程序设计的。这样在系统中集成传统加速器以

及 C-core 后，程序的并行部分可以在传统并行加速器上执行，而串行部分可以在 C-core 上执行，这样将大大提升专用协处理器执行程序的代码覆盖率并提高整个系统的能量效率。

C-core 自动生成技术是 CoDA 架构实现的工程基础。采用自动生成的方式就可以在可接受的时间内生成大量专用协处理器并使用它们覆盖应用大部分的执行。这样才能使得系统的能量效率明显提升。

C-core 技术研发之后，本团队又进行了 GreenDroid 项目的研究工作。GreenDroid 解决了哪些软件需要被硬件专用化并提出了将 C-core 集成到通用架构的简单方法。此外，GreenDroid 还分析了面向单个应用程序专用化，可以获得的能量效率提升。至此，本团队完整地论证了 CoDA 在工程实现上的可行性以及可以获得的能量效率优势，之后提出了 CoDA 架构。

除了本团队的研究之外，工业界的半导体发展路线图 ITRS 在 2012 年将 Dark Silicon[1] 以及 CoDA 列入了关于低功耗设计技术路线图中（2013 年发布），见图 1-5[2]。另外本团队 CoDA 架构的研究也受到美国国防部先进研究计划局(DARPA)通过下属未来架构研究中心(Center for Future Architectures Research, C-FAR)给予研究经费支持。

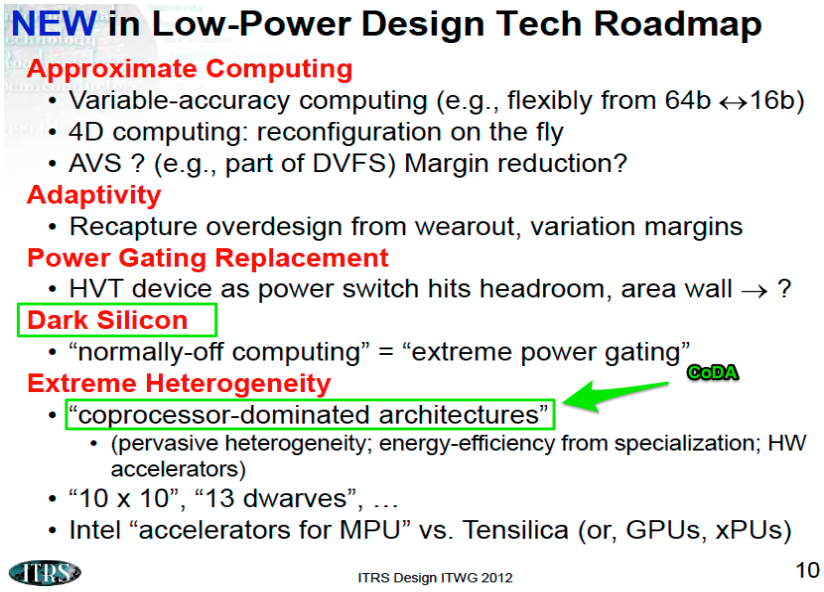


图 1-5 ITRS 将 CoDA 列入新的低功耗设计技术路线图[15]

1.2.2 课题的提出

前面的工作证明在单个应用的系统或较小负载的系统中使用专用协处理器可以带来更好的能量效率。但是常见的系统上通常运行着大量不同的负载，CoDA 架构也需要提高这类复杂应用环境下系统的能量效率，也就是说架构师需要为几十或上百个不同应

[1] ITRS 在这个报告中用 Dark Silicon 来表示一类使用 power off 技术使得硅成为暗逻辑的技术。
[2] ITRS 2012 年的预测报告是在 2013 年发表，由于 2013 年的预测报告在本论文撰写时还没完全整理发表，所以 2012 的报告为目前可见的最新预测报告。
http://www.itrs.net/Links/2012Winter/1205%20Presentation/DesignSD_12052012.pdf

用负载设计专用协处理器，并将这些专用协处理器集成到 CoDA 架构。这使 CoDA 架构面临一些理论和工程方面的挑战。这些挑战来自于不断向 CoDA 架构中集成更多的专用协处理器所带来的下列问题。

首先，随着 CoDA 架构规模扩大，集成大量专用协处理器所带来的能量开销也随之增加，这些开销吞噬了 CoDA 所带来的潜在能量优化。尽管与通用处理器相比，每一个单独的专用协处理器都可以带来性能或能量优化，但是集成大量专用协处理器到一个架构中会使片上互连更为复杂，同时也需要使用更为复杂的存储系统。

其次，尽管与传统设计相比，CoDA 架构中在任意时刻芯片上大部分晶体管会处于空闲状态（例如：门控状态），但是由于芯片规模较大，这些空闲的晶体管所带来的漏电功耗也会给设计带来很大问题。

第三，增加专用协处理器的数量会增加程序在他们之间跳转的频率、增加跳转开销（互连更复杂，延时会增加）并会影响 Cache 的性能。如果设计者没有仔细对这些因素进行权衡，这些无用的开销会吞噬大量使用专用协处理器所带来的能耗或性能优势。

第四，随着集成专用协处理器数量的增多，手工集成这些处理器不但耗时而且容易出错；另外，由于 C-core 中大部分电路都是受到多时钟周期路径约束，这种电路是否可以用现有 EDA 工具实现并且正确运行都需要工程实现探索。

上述这些挑战的实质问题是需要探索解决 CoDA 架构的可扩展性问题，该问题是 CoDA 架构的核心问题。CoDA 架构的可扩展性问题包括以下几个方面：

1）CoDA 所支持应用的可扩展性。由于 CoDA 架构中集成的都是专用协处理器，每一个专用协处理器都是面向特定应用生成的。因此这种架构是否适用于当今的应用程序是一个值得研究的问题。只有当 CoDA 架构可以适用于大规模、多样的应用程序，它才值得被研究。支持应用的可扩展性体现在 CoDA 对大规模应用的适用性上。此外，CoDA 架构需要尽量维持编程模型来简化应用程序移植。本文第二章研究这个问题。

2）CoDA 架构的可扩展性。因为 CoDA 将要集成成百上千的专用协处理器，这就需要 CoDA 硬件具有可扩展的架构。由于应用负载多种多样，呈现不同特征，这要求 CoDA 架构的扩展可以在不同维度上适应应用负载，因此 CoDA 硬件架构要具有多个维度的可扩展性。此外，为了适应暗硅时代，CoDA 架构还需支持多种功耗管理技术并且架构、功耗管理技术和程序执行策略要紧密结合。本文第三章解决这个问题。

3）CoDA 能量效率角度的可扩展性，这是 CoDA 架构可扩展性研究的核心问题。CoDA 的提出就是为了追求高能量效率。在互连、存储和漏电功耗等开销的影响下，大规模 CoDA 架构的能量效率与通用架构相比是否还可以成倍提升，是关系 CoDA 设计是否可扩展的核心问题。只有大规模 CoDA 具有高能量效率，CoDA 架构扩展才是有效的，才能证明 CoDA 适应暗硅时代芯片特点。这是本文第四章设计空间探索的重点。

4）CoDA 工程实现角度的可扩展性。硅是检验结构设计的唯一标准[48]，对于新提出

的架构需要更加谨慎。为了证明所提的 GreenDroid/CoDA 架构可以使用现有开发工具、脚本和工艺实现，需要进行 FPGA 系统原型设计实现以及流片工作。只有可以工程实现的新架构才是有价值的。本文第五章介绍 CoDA 工程实现方面的工作。

此外 CoDA 架构也会带来专用协处理器对应用负载动态执行覆盖率以及使用模型方面的问题。传统的专用协处理器仅仅支持一部分关键的应用（例如：视频解码、加密解密），而在 CoDA 架构中几乎所有的应用都需要使用专用协处理器，并发执行的多个应用可能会同时使用多个专用协处理器。如果应用程序竞争使用某些特定的专用协处理器，竞争失败的线程要么等待直到专用协处理器空闲要么返回到通用处理器上执行，这样性能和/或能量效率将受到影响。对于多线程负载，这些冲突会显著地降低 CoDA 的效率。这种影响到底有多大，是否可以缓解也需要进一步仔细的研究。

其他一些需要讨论的问题还包括：a）大规模的 CoDA 设计由于会使用更多的晶体管，这样是否会对芯片加工的良率产生影响；b）大规模的 CoDA 架构在应用负载执行时，与通用架构相比是否对片外访存带宽产生更大的压力；

1.3 本文的工作和创新点

本文的主要工作是作者作为西北工业大学和 UCSD 联合培养博士研究生在 UCSD GreenDroid 项目组和 Dark Silicon 研究中心工作三年完成。在 C-core 和 GreenDroid 解决了面向单个应用时使用专用协处理器可以带来数倍的能量效率优化之后，本文探索了面向大规模应用集成成百上千专用协处理器的 CoDA 架构是否可扩展并具有较高能量效率的问题。除了研究工作，作者作为 UCSD GreenDroid 项目系统设计负责人，领导了 GreenDroid/CoDA FPGA 原型系统的硬件设计、编写了 Linux 驱动、修改并重新编译了操作系统、优化了部分 C-core 硬件结构并为 2 款将要流片的芯片准备 RTL 代码以及完成 45nm 前端设计和验证工作。作者在西北工业大学学习期间作为主要架构师设计了基于粗粒度可重构阵列的众核架构流处理器，作者独立设计并实现了该处理器软件管理的存储系统并编写了管理存储器的应用程序 API 接口工具包。本文结合作者博士研究生阶段完成的实际工作，主要研究和工程工作包括以下几个方面。

（1）支持流编程模型的 S-core 协处理器。在本文介绍的 C-core 支持程序中的非规则应用之后，对于程序中具有流特征的特定功能函数、数据并行的规则部分提出了 S-core 流处理器。S-core 的存储系统设计参考了斯坦福大学 Imagine 处理器和国防科大的 MASA 处理器，作者对微结构进行了设计和实现；S-core 的计算部分使用了粗粒度可重构阵列，通过重构支持特定功能。此外，为了简化程序员编程，作者为软件管理的存储器设计了驱动工具包，并封装了一系列用于申请、释放、传输、分拆合并流数据的 API 函数。

（2）以浏览器为例研究了 CoDA 对应用程序的适用性。在团队其他成员分析了安卓系统之后，以浏览器作为特例进一步详细分析了安卓应用程序的执行。重点分析了浏览器访问网站时执行的静态指令数量、线程情况、代码共享以及函数调用关系等等信息。

使用这些信息以及 GreenDroid 项目前期对专用协处理器面积的评估结果，估算了覆盖90%浏览器执行所需专用协处理器的面积以及数量，并且这些硬件化的代码大部分都是被其他应用所共享的。

（3）建立了 CoDA 架构分析评估模型。本文在团队其他成员建立的时序精确的功能仿真器的基础上，进一步建立了 CoDA 架构的评估模型。该模型可以分析特定 CoDA 设计的面积、性能和能耗。分析评估模型中所建模的 CoDA 架构在硬件结构上具有多维度可扩展的特点，并支持多种能耗管理技术。能耗管理技术、架构和程序调度紧密配合是 CoDA 架构的设计要点。分析评估模型中的设计空间参数包括一些顶层的架构参数，例如瓦片的面积、Cache 的配置等；还包括一些实现方面的参数，例如选取的晶体管工艺库、采用的功耗管理策略等等。

（4）使用分析模型评估了 7200 种不同的 CoDA 设计。这些 CoDA 架构不但包括面向小型负载的具有几个专用协处理器的架构，还包括面向大型负载的集成 352 个专用协处理器的复杂 CoDA 架构。分析模型评估了每种 CoDA 设计的面积、性能和能耗信息，并找到支持特定应用规模的最优 CoDA 设计。通过分析最优 CoDA 设计发现 CoDA 扩展时各个子部件能耗的变化趋势，以此指明未来 CoDA 设计所需要注意的问题。此外还对 CoDA 与芯片良率以及 CoDA 对片外访存压力进行了讨论。

（5）建立了 GreenDroid FPGA 原型系统。本文作者作为 GreenDroid 团队系统设计负责人建立了最早的 GreenDroid FPGA 原型系统。原型系统的基础是 Basejump，它为设计提供了外设控制器、跨多芯片能力、驱动程序、PCB 板以及芯片封装等等，这些设计不但被本团队使用而且也开源给其他团队使用。本文介绍了作者负责完成的 Basejump FPGA 系统实现工作。GreenDroid 原型系统就是将 GreenDroid 设计集成到 Basejump 搭建完成的。本文还介绍了部分芯片设计工作。作者还详细阐述了工程设计中所使用的设计方法学：模块大量复用、设计参数化以及贯穿始终的自动化，这是我们 4-6 个研究生团队完成这个项目的关键。

本文研究紧紧围绕上述内容展开，进行了以下几个方面的创新性工作：

（1）研究了 CoDA 对应用的适用性，并以此说明 CoDA 适合暗硅时代。本文分析了安卓移动软件栈，发现大部分应用是基于共享原生库和虚拟机的，硬件化这部分软件就可以使得应用的大部分运行在专用协处理器上。之后重点分析了安卓浏览器，并使用硅构造专用协处理器实现了这个浏览器。实验结果表明在 22nm 工艺下 7mm^2 的硅面积用于构造专用处理器就可以覆盖浏览器 90%的运行。使用可接受的硅面积就可以覆盖应用执行，证明了 CoDA 架构适合暗硅时代。

（2）针对快速探索 CoDA 设计空间的需求，提出了 CoDA 架构分析模型，并对本文提出的多维度可扩展 CoDA 架构进行建模。该架构可以由不同数量的瓦片组成，每一个瓦片可以包含不同数量的函数粒度专用协处理器，并且每一个专用协处理器都可以是

异构的。分析模型用来评估每一种特定 CoDA 架构的能量、面积和性能；模型参数既包含了高层次的体系结构参数，也包含低层次的电路实现参数。

（3）探索了 CoDA 架构在不同 Cache 配置、瓦片大小、粗粒度能耗管理策略以及晶体管实现等参数下的能量效率问题。在最优化的参数条件下，与通用架构相比小规模 CoDA 设计可以带来 5.3 倍的能量效率优化和 5 倍的能量延时积（energy-delay product, EDP）优化；而对于支持上百个应用的大规模 CoDA 设计，可以带来 3.7 倍的能量效率优化和 3.5 倍的 EDP 优化。这说明为大规模应用而设计的大规模 CoDA 扩展是有效的。此外，本文发现 CoDA 设计即使采用了激进的能耗管理策略，漏电功耗所占总功耗的比例仍然随 CoDA 规模增大而增大。

（4）探索了并发执行对 CoDA 能量效率的影响。积极的影响是这些同时运行的程序或线程可以分摊漏电功耗等固定的开销，这样可以提高系统的能量效率。消极的影响是，当驱动 CoDA 生成的目标应用集合和实际运行的应用集合不匹配时，会造成大量程序竞争某些专用协处理器，系统的平均能量效率将大大降低。本文提出 CoDA 架构集成覆盖多个函数功能的融合 QsCore 来减少竞争冲突。实验表明使用融合 QsCore 的方式，仅仅增加 41%的面积就可以提供 2 倍数量的专用协处理器，并使得非均匀分布负载的能量效率提高 11.1%~22.1%。

（5）针对使用当前工艺实现的 FPGA 模拟下一代工艺实现的 CoDA 芯片时，单个 FPGA 芯片资源不足的问题，提出了跨多芯片可扩展的 2D-mesh 片上网络。该网络由跨芯片的环形网络连接，并为跨芯片的每一个 2D-mesh 物理通道分别提供跨芯片的流控机制。跨芯片的环形网络提供了 ASIC 芯片到 FPGA 以及 FPGA 之间两种可选连接方案。通过使用该设计方案，本文使用两块 Virtex 6 FPGA 芯片首次实现了 CoDA 架构原型系统。

1.4 论文的结构

本文围绕着解决使用墙问题和适应暗硅时代所提出的 CoDA 架构展开，重点关注了 CoDA 架构理论研究和工程实现所遇到的挑战，全文共分为六章。

第一章为绪论，介绍了课题的研究背景和学术界、工业界的相关研究工作，简述了课题的主要研究内容、创新点和组织结构。

第二章介绍了 CoDA 研究的工程基础以及 CoDA 对程序的适用性。CoDA 研究的工程基础包括函数粒度的可自动生成专用协处理器 C-core 以及支持流编程模型的 S-core。之后讨论了 CoDA 架构对应用程序的适用性。首先分析了安卓系统，之后针对安卓系统上最重要的应用浏览器进行了重点分析，发现 CoDA 架构适合于这种应用环境。这些为 CoDA 研究提供了工程基础和理论基础。本章的内容来发表于论文“GreenDroid: An architecture for dark silicon age”、“SiChrome：A Silicon Browser”和“An Evaluation of the Many-core Longtium SP Computer System”。

第三章描述了 CoDA 的架构、能耗管理策略、程序执行策略、应用负载、分析模型建立以及模型的参数空间。因为 CoDA 架构十分复杂，体系结构、工艺实现等可选参数众多，无法真实使用电路实现每一种设计并进行评估，所以本文在第三章中建立了针对 CoDA 的能量、性能和面积的分析模型。这个模型是探索 CoDA 能效问题和并发执行对 CoDA 影响的基础。该章的主要内容发表于论文“Exploring Energy Scalabiligy in Coprocessor-Dominated Architectures for Dark Silicon”。

第四章探索了 CoDA 在规模逐渐扩大之后，在能量效率角度是否可扩展并且也讨论了并发执行对 CoDA 架构能量效率的影响。该章也对大规模 CoDA 对芯片加工良率和片外访存的影响进行了讨论。该章主要内容发表于论文“Exploring Energy Scalabiligy in Coprocessor-Dominated Architectures for Dark Silicon”。该章的部分内容发表于论文“Tolerating memory latency: L2 cache actively push architecture”和“AAP and AAPM: improved prefetching structures of the L2 cache”相关。

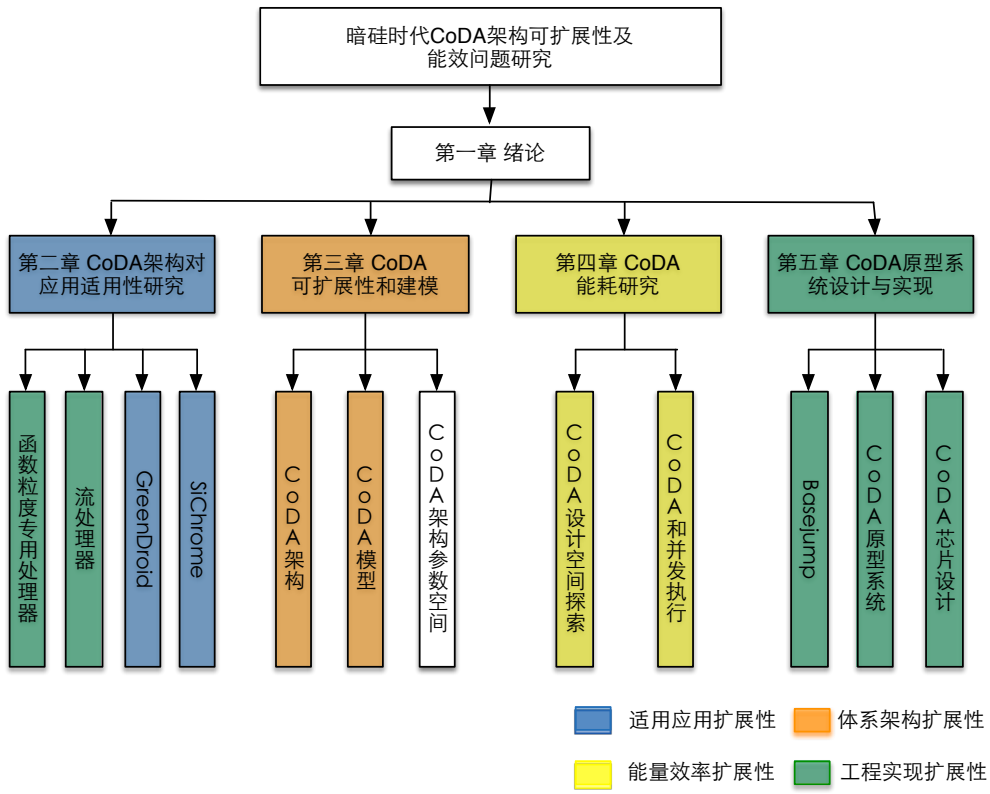


图 1-6 本文研究内容的组织结构

第五章探索了 CoDA 设计的工程实现。该章首先提出了辅助快速构建原型系统的 Basejump；之后介绍了小规模 CoDA 设计 GreenDroid 系统 FPGA 实现以及验证工作；最后探索了简单 CoDA 设计的芯片实现。本章内容是对 GreenDroid/CoDA 架构硬件工程实现的总结和感悟。

第六章总结了本文的工作，并展望了下一步的研究和工程工作。

论文的结构框图如图 1-6 所示。

2 CoDA 架构对应用适用性研究

CoDA 架构的提出是为了寻找适应暗硅时代芯片资源特点的新型体系结构。CoDA 架构提出的基础除了未来芯片上大量的暗硅资源外，还包括自动生成的专用协处理器以及软件的编程特点。本章介绍 CoDA 架构的工程基础并讨论这种架构对应用的适用性。本章首先介绍函数粒度自动生成的专用协处理器 Conservation Core(C-core)。与传统支持规则并行程序的加速器不同，C-core 支持程序中的非规则代码；其次介绍支持流编程模型的 S-core，它由两大部分组成软件管理的存储器和粗粒度动态可重构阵列。S-core 可以用来覆盖程序中具有流数据特点的并行部分，第五章对 S-core 进行了芯片设计；之后，通过 GreenDroid 和 SiChrome 论述了 CoDA 架构对应用的适用性，实际上也是 CoDA 支持的应用是否可扩展[1]。本文分析了安卓系统的架构，发现这种架构较为适合专用化，并以浏览器为例评估了硬件化浏览器所需的资源。评估发现使用可接受的硅面积就可以覆盖应用执行，证明了 CoDA 架构适合暗硅时代。

2.1 函数粒度专用协处理器 C-core

函数粒度的专用协处理器 C-core 由于是从应用程序源代码中自动生成的，并支持任意的非规则代码，所以可以覆盖应用程序的大部分代码也就是可以使用这个技术将应用的大部分硬件化。因此 C-core 成为 CoDA 研究的基础，本小节详细介绍 C-core 的技术细节。首先介绍 C-core 的发展历程以及自动生成 C-core 的工具链；其次介绍函数粒度 C-core 的硬件结构；再次介绍如何在通用架构中集成 C-core；之后介绍 C-core 的跳转执行过程；最后介绍 C-core 的评估结果。

2.1.1 C-core 自动生成工具链

随着对自动生成专用协处理器技术研究的不断深入，C-core 生成技术也在不断发展。其发展至今最重要的三个阶段：1）2010 年 ASPLOS 会议上提出使用墙问题[5]，并提出了 Conservation Cores（C-core）技术，此时的 C-core 主要目标是提高能量效率，结构也较为简单，都是直接从编译的数据通路以及控制通路转换而来的。2）2011 年的 HPCA 会议上提出的选择性去流水线技术(selective Depipelining, SDP)以及 Cachelet 技术[13]，对 C-core 进行了较大的优化，这两个技术可以同时提高 C-core 的性能和能量效率。本文主要内容的研究和工程实现大体上基于这个版本的 C-core。3）2011 年的 MICRO 会议上提出的 QsCores 技术[19]，可以为功能相似的软件代码生成一个专用协处理器，从 C-core 的角度可以认为将功能类似的多个 C-core 融合在了一起。这将有效的减少专用协处理器面积/数量，进一步提高能量效率，本文第四章使用了这种 C-core。

除此之外，团队其他成员还对生成 C-core 的工具链进行了较大优化，从早期使用多

[1] 应用可扩展是指 CoDA 可以支持大量不同的应用，并且所需的芯片资源适度。

种不同的编译工具（GCC、LLVM、CodeSurfer[49-51]、OpenIMPACT[52]以及 Raw-GCC[53]），转变为仅仅使用LLVM[1]一种编译器工具。对工具链生成的C-core其他方面的优化包括：1）主处理器和 C-core 之间的互连从早期的扫描链变成了树形网络；2）进行了运算器融合的优化，例如乘法器和加法器被优化为乘加器；3）对串行的多个运算操作，如果满足交换律就进行二叉树平衡优化，提高 C-core 性能；4）尝试自动用硬件实现软件流水线(software pipelining)等等。

C-core 的硬件电路是使用团队设计的自动化工具链从软件源代码中转换成硬件的。工具链运行的第一个步骤就是对应用负载进行分析，识别出热代码和冷代码。热代码通常是某些函数、循环或者一些基本块，这些代码占用了大部分程序运行时间并消耗了大部分的能量，这些热代码将被工具链转换为硬件的 C-core。识别出热代码之后，工具链使用 LLVM 自带的 Clang 编译器前端工具将 C/C++代码转变为 LLVM 中间码(IR)。为了更好的对软件和电路进行优化，还设计了更为适用的中间码格式(C-Core IR 和 Arsenal IR)，并编写了大量的 LLVM 语法分析器对软件代码（例如循环展开、公用表达式删减等等）和硬件电路进行优化（例如，在非关键路径上使用能量效率更高的低速运算器等）。最终 LLVM 生成 C-core 的硬件 Verilog 代码、硬件工具使用的时序约束文件以及时钟精确的软件仿真代码。工具链将生成的软件仿真代码加入 MIT Raw 处理器的仿真器 BTL 中进行软件的仿真。生成的硬件代码和约束文件使用脚本处理后可以自动集成到 GreenDroid/CoDA 处理器的开发流程中。

2.1.2 C-core 硬件结构

函数粒度的专用协处理器 C-core 的硬件实现由一系列的程序基本块硬件电路拼接而成。为了兼容程序二进制调用接口 ABI，C-core 的接口处添加电路来模拟通用处理器使用的通用寄存器和栈结构。这样一个 C-core 就实现了一个软件函数的功能。

图 2-1 展示了 C-core 中一个基本块的电路，包括了数据通路、控制逻辑、快慢时钟以及存取逻辑。图中的 C-core 是由右下角的 C 代码经过工具链自动生成。图上部的快时钟（fast clock）用于控制 C-core 访存逻辑(load 和 store)以及控制流图生成的状态机，这个时钟的频率和主处理器时钟频率相同；慢时钟用于控制基本块与基本块之间插入的寄存器也就是图示电路两端的寄存器。慢时钟本质上是一个脉冲信号，每一个基本块的慢时钟两个脉冲信号之间的拍数都不相同，具体的数值由工具链中的调度器根据路径中的各种运算器件延时计算得出。如图所示，为了降低能量消耗以及提高 C-core 运行速度，基本块内部的逻辑除了 load 操作之后有寄存器外，其他的运算单元后面的寄存器被全部移除。图中椭圆形区域展示了控制逻辑(CFG)，控制逻辑的硬件由状态机构成并在快时钟的控制下工作。对于与存储交互的操作，状态机中将生成带有循环的状态，这个状态

[1] 之所以选择 LLVM 是因为该编译器的设计基于模块化的思想，可以很容易的增加不同的优化和对代码进行变换，甚至有很多开源的优化代码可以参考。另一方面，LLVM 目前越来越流行，由苹果公司和伊利诺伊大学香槟分校共同支持，并且目前已经是苹果电脑默认的编译器。

可以使状态机等待存储操作完成之后才进入下一个状态。

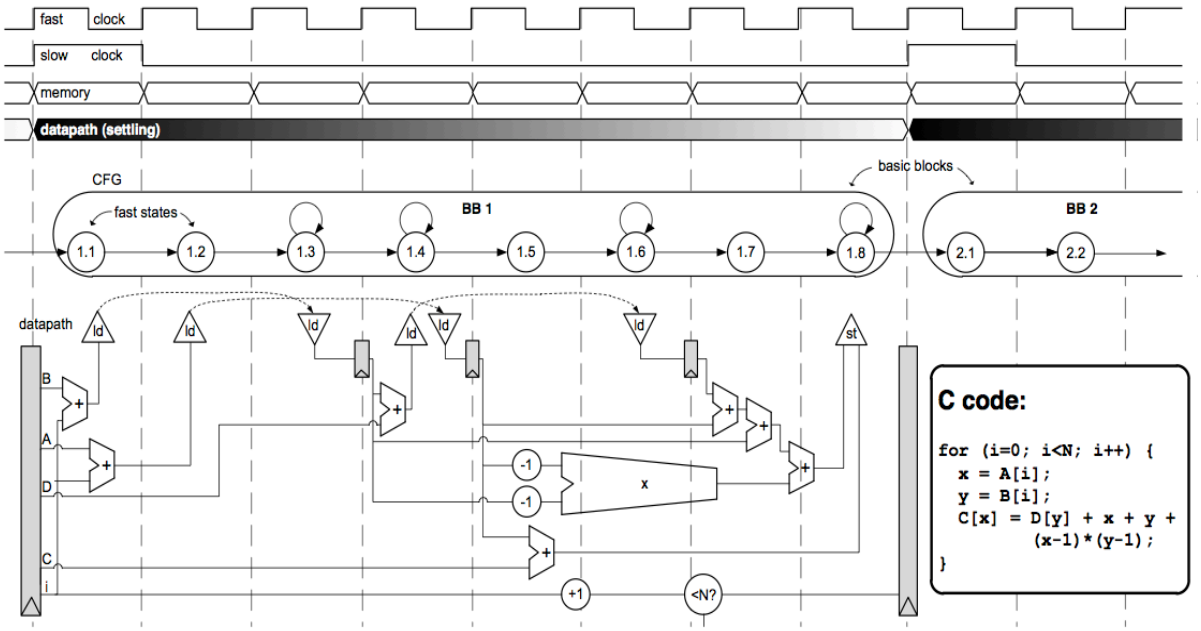


图 2-1 C-core 中基本块结构图

整个 C-core 就是由一组类似图 2-1 中所示的基本块逻辑拼装而成，C-core 中整体的控制逻辑也是根据程序在基本块之间的跳转情况由基本块中的状态机拼装而成。此外，为了满足程序的二进制调用规则（ABI），设计需要在 C-core 中添加一些寄存器和逻辑来满足这些规则，这些电路在 C-core 中被打包成一个特殊的基本块。整个 C-core 内部除了 load 和 Store 指令外，其他软件指令全部用专用硬件实现，C-core 内部不存在译码逻辑。

通用处理器架构集成大量 C-core，需要所有 C-core 具有统一的接口。大体上接口分为两部分：首先是通用处理器核与 C-core 的接口，这个接口主要用于主处理器向 C-core 传递参数、指针等等数据或者从 C-core 读取结果或者状态等。另一部分是 C-core 与存储器的接口，这个接口主要是为了使 C-core 可以读写存储器。主处理器核与 C-core 之间数据的传输通过一组树形网络，树形网络的地址编码包括三部分：1）C-core 编号；2）C-core 中的基本块编号；3）该基本块中的寄存器编号。通过这组树形网络，主处理器就可以读写 C-core 内部的每一个寄存器。这样使得 C-core 也支持异常，并且通过 patch 的机制（参数通用化、运算器通用化等等）支持未来版本的软件。为了使 C-core 对程序员透明，也就是程序员不需要修改原来运行在主处理器上的软件源代码，需要主处理器和所有 C-core 共享数据 Cache。主处理器、C-core 与共享数据 Cache 之间也采用树形网络。为了维护 Cache 一致性，瓦片内部同一时刻仅仅允许激活一个处理器，或者是主处理器或者是某一个 C-core。详细的信号列表见表 2-1。

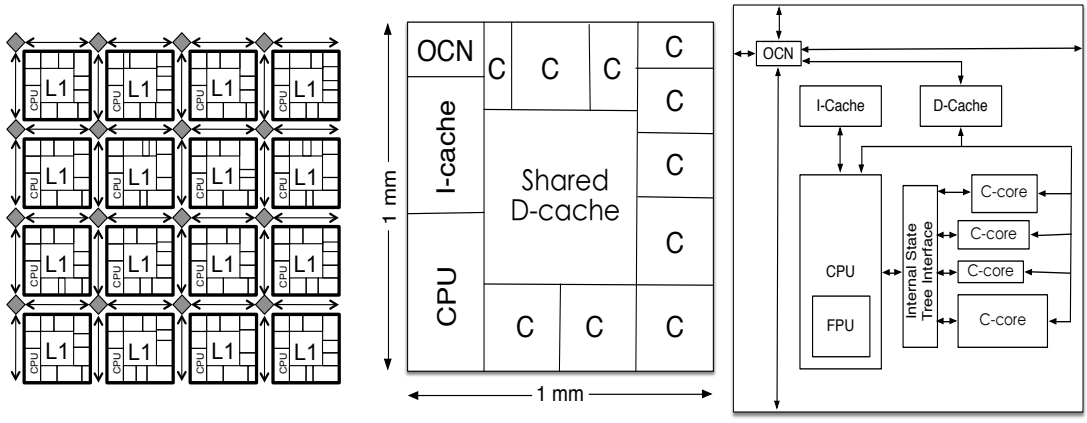
表 2-1 通用架构集成 C-core 的接口信号

名称	位宽	方向	连接模块	描述
tree_addr	32	In	主处理器核	树形网络访问地址
tree_store_value	32	In	主处理器核	存储到 C-core 中寄存器的值
tree_re	1	In	主处理器核	读寄存器信号
tree_we	1	In	主处理器核	写寄存器信号
tree_load_value	32	Out	主处理器核	读取 C-core 中寄存器的值
clk	1	In		时钟信号
reset	1	In		复位信号
mem_valid	1	In	共享 Cache	Cache 取回的数据有效
mem_load_value	32	In	共享 Cache	Cache 取回的数据
mem_addr	32	Out	共享 Cache	访存地址
mem_store_mode	2	Out	共享 Cache	访存模式，0 为 8 位，1 为 16 位，2 为 32 位
mem_store_value	32	Out	共享 Cache	写 Cache 操作的写回值
mem_access_type	2	Out	共享 Cache	访存类型；2 为读取。1 为写回，0 为无操作
attention	1	Out	主处理器核	类似中断信号,通知主处理器核 C-core 需要处理
err_flag	32	Out	主处理器核	错误信号标志,通知主处理器核 C-core 哪里执行出现错误

2.1.3 集成 C-core 的架构

为了向系统中集成大量的专用协处理器，需要选择具有可扩展性的架构。MIT 的 Raw 处理器由于采取了瓦片化的设计并且使用的 2D-mesh 片上互连网络理论上可以无限扩展，就成为了较为理想的基础架构。图 2-2 是 C-core 集成到 GreenDroid 中后整个系统的架构。其中图 2-2(a)是整个 GreenDroid 的结构框图，GreenDroid 采用了 2D-mesh 互连的瓦片化架构。每一个瓦片的布局如图 2-2(b)，其中包括一个通用主处理器(MIT Raw)、通常包含 8-15 个 C-core、指令 Cache、被主处理器和 C-core 共享的数据 Cache 以及网络接口(on-chip network, OCN)。图 2-2(c)是瓦片内部各个模块的互连关系。架构从两方面来保证不修改编程模型，也就是程序员不需要修改源代码就可以移植程序到 C-core 上运行。首先，C-core 兼容程序调用二进制接口 ABI；其次，主处理器和 C-core

共享 Cache。共享 Cache 就避免了数据在不同存储器之间迁移，避免程序员手工修改源代码并使得程序员负责数据调度。



(a) GreenDroid 架构　　(b) 瓦片内部结构　　(C) CPU 和 C-core 互连

图 2-2 集成 C-core 的 GreenDroid 架构

2.1.4 C-core 跳转执行

生成 C-core 的工具链既会生成 C-core 的硬件电路和软件的仿真器，还会对生成的可执行程序进行少量修改。在可以调用 C-core 的地方添加查看 C-core 状态的代码，使得在执行时可以动态检查 C-core 的状态。这样如果 C-core 空闲就跳转到 C-core 执行程序，如果 C-core 忙就让程序在主处理器上运行或者等待。

如果程序决定跳转到 C-core 执行，主处理器就将程序运行所需的参数、栈指针、全局变量指针等传递给 C-core。当程序运行结束之后，主处理器将结果从 C-core 中读回。

C-core 的调用兼容了 MIT Raw 处理器的 ABI，Raw 处理器的 ABI 与 MIPS O32 ABI 的约定十分类似。对 ABI 的兼容包括两个部分：一个是参数的传递，另一个是栈的使用。Raw ABI 约定寄存器$v0、$v1 用于执行结果返回，寄存器$a0-$a3 用于传递参数，此外有专门的寄存器用于传递 SP（栈指针）和 GP（全局变量指针）。设计的 C-core 中实现了这些寄存器，调用 C-core 时通过主处理器与 C-core 的硬件接口把值写入这些寄存器。对于每一个全局变量，在 C-core 中要实现一个地址偏移寄存器，这样 C-core 运行时就可以通过 GP 寄存器和偏移寄存器计算出全局变量地址，之后发送读写请求来操作这些全局变量。与软件的函数调用栈不同 C-core 硬件的栈中没有局部变量和临时变量，这些变量的值存储于 C-core 的硬件流水线中。

当获得函数执行所需要的所有数据后，C-core 电路会根据生成的硬件状态机进行执行。当 C-core 结束执行后，会置位 attention 信号，此时主处理器将 C-core 运行结果从 C-core 读出。

除了叶函数的执行，C-core 也支持直接调用其他函数以及递归调用，这些都需要对 C-core 的硬件栈结构进行添加。

为了更形象地说明程序如何在主处理器和 C-core 之间跳转执行，本小节列举一个应

用程序函数级别的“伪代码”，并演示如何跳转。如图 2-3 左侧所示，一个叫做 Hello C-core 的应用调用了 3 个函数，工具链分别为这三个函数生成了 3 个 C-core 并集成到通用架构中。这样当应用执行到函数 1 的时候，就跳转到 C-core_f1 上执行，结束后跳转回通用处理器；其他函数的执行也遵循同样的规则，过程见图 2-3 右侧。

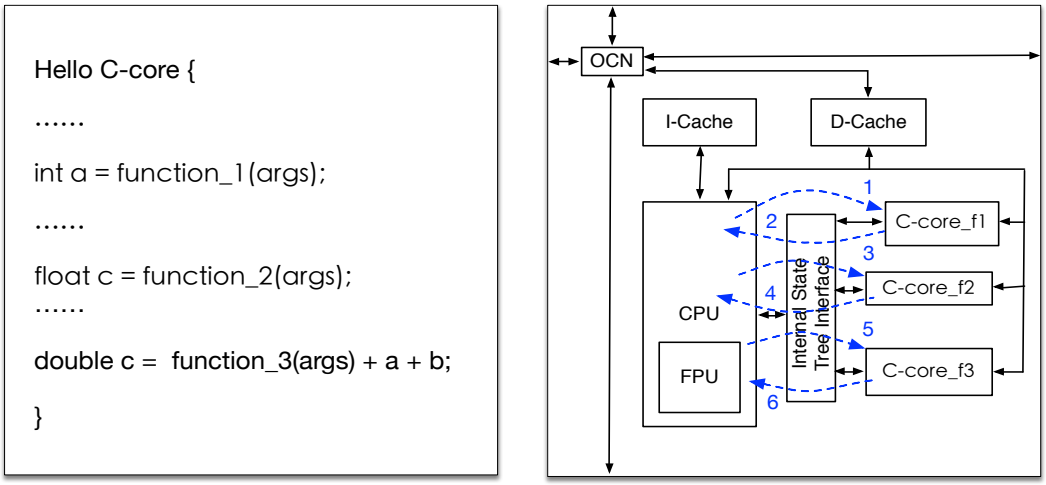


图 2-3 应用函数级伪代码及主处理器和 C-core 间跳转执行

2.1.5 C-core 评估

本小节介绍了 C-core 的结构、工具链、C-core 的接口以及调用方法。现在介绍使用 C-core 后，系统的能量效率提升情况。图 2-4 取自于论文[13]，是对系统中运行的整个应用程序进行性能和能耗统计的结果。图中的 In-SW 指所有程序都运行在主处理器上并且系统没有集成 C-core。本文主要介绍和其他章节使用的是 C-core 第二个优化版本，即图中的 ECOcore。三张图中最上面的是能量延时积，中间的图是程序运行时间，最下面的图是能量消耗的分布情况。使用 ECOcore 可以使程序运行时间最多减少 33%并且与通用处理器核相比运行时间都有所降低。平均能量延时积优化可以达到 59%，这些优化的取得大部分依赖于对能量消耗的优化，通过计算可知在运行完整的测试程序时对能量的优化率最高可以达到 70%。

论文[13]中的通用主处理器采用 45nm 工艺实现的 MIPS 24KE 处理器，并假设该处理器工作时钟为 1.5GHz，数据 Cache 和指令 Cache 的性能和功耗数据来自于 CACTI5.3。专用协处理器 ECOcore 的评估采用工具链生成的 Verilog 文件，并使用 Synopsys DC(C-2009.06-SP2)进行前端设计、IC Compiler(C-2009.06-SP2)进行后端设计。评估面向的工艺为 TSMC 45nm GS 工艺库。

此外需要说明的是类似 C-core 这种结构的新型专用协处理器核，优化空间巨大，并且很多优化都是可以同时优化性能和能量效率。例如前面提到的融合不同的运算器、对过长位宽的运算器进行缩减(例如对 4 位加法用 4 位加法器替代 32 位加法器)等等。未来对 C-core 结构的不断优化，可进一步提升这类专用协处理器核对能量效率的优化效果和运算性能。

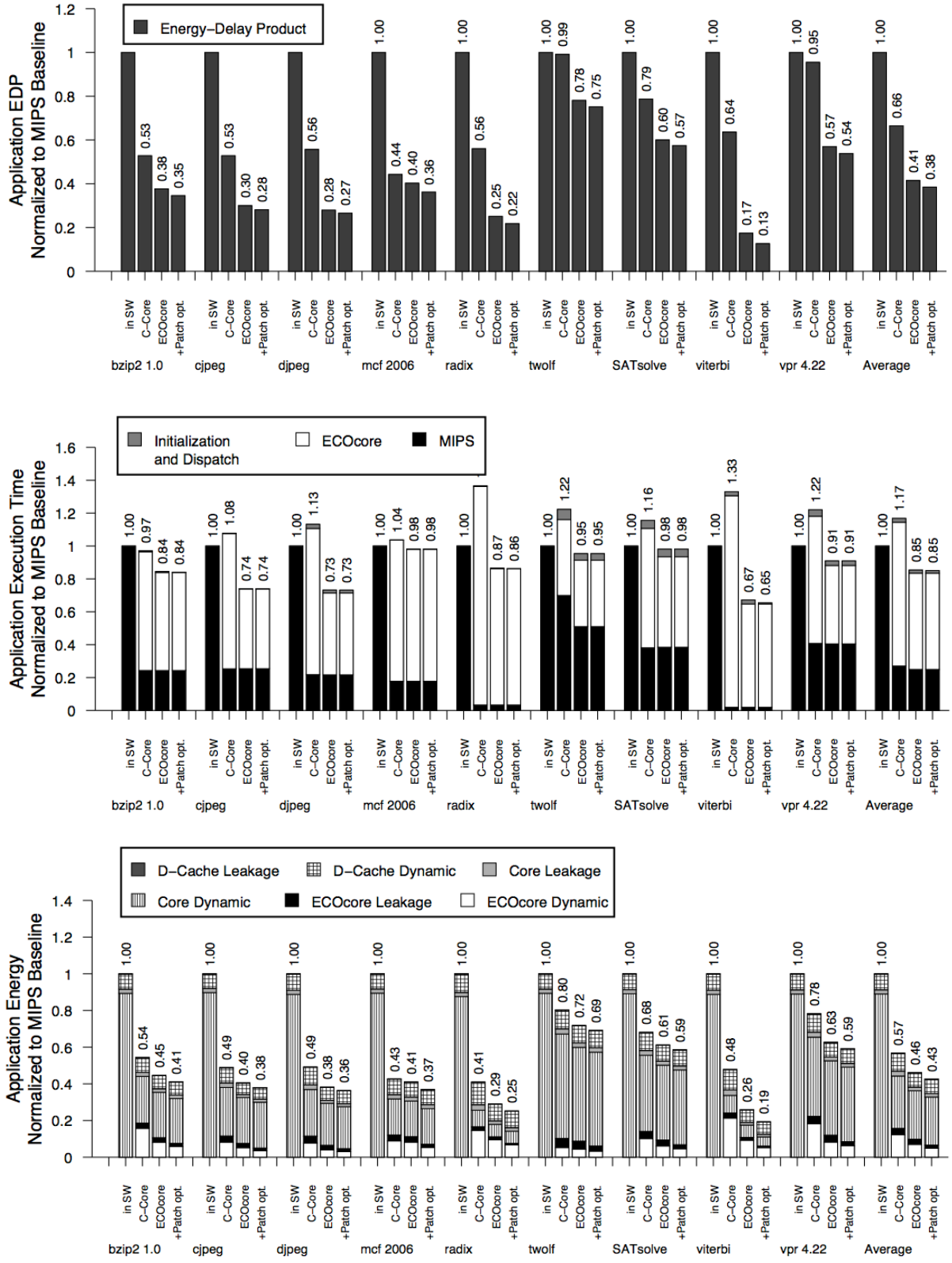


图 2-4 应用程序性能和能量效率

2.2 支持流编程模型的 S-core

在 C-core 较好地匹配非规则代码（例如包括大量分支和跳转指令的代码等）的基础

上，本小节针对规则代码提出了 S-core 架构。为了高能量效率的处理并行计算，团队需要为 CoDA 体系结构加入适合并行计算的专用协处理器。文献[21]尝试向系统集成了可重构阵列[1]并通过将控制复杂代码和并行代码映射到阵列上来提高系统的性能和能量效率。本小节提出的 S-core 也以粗粒度可重构阵列为基础，试图覆盖具有多种并行性的应用代码，例如数据并行、指令并行、线程级并行等。此外，针对文献[54]发现的频繁重构导致系统性能和能量效率损失的情况，本文认为仅仅使用重构阵列覆盖计算重构比较高的程序部分就可以解决这个问题，其他程序并行部分可通过向系统中集成 SIMD、VLIW 等类型的加速器进行覆盖。

此外，流处理器作为一种新型的架构，不仅适合具有多种并行性的应用，而且还是一种典型的高能量效率架构[55-58]。此外，流处理器所支持的流编程模型中的核心程序就具有高计算重构比的特点。因此本文设计的 S-core 也支持了流编程模型，并具有典型的流处理器架构特点。

2.2.1 流和流编程模型

为了说明流编程模型，首先要知道什么是流和流处理的特点。流是不间断的、连续的、移动的记录队列，队列长度可以是定长或不定长的[59-62]。这些记录可以是简单的向量元素，也可以是较为复杂的图像像素。这种类型的数据在使用过程中是非常容易预测的，并且有较好的生产者-消费者局部性。与传统架构不同，流处理有以下几个特点：计算和存储解耦合，计算过程分解，显示通信，并行处理等特点[59-62]。

为了满足流处理的存取和计算特点，人们提出了流编程模型。其核心思想就是将应用组织成流程序和核心程序，以便挖掘应用本身的并发性和局部性。这种流级（StreamC）和核心级（KernelC）的两级编程模式，分别对应流的组织调度以及对流记录的计算，通过这种方式就可以将应用程序划分为流级程序及多个核心程序，如图 2-5 所示。核心程序是一个计算密集型的函数，对流中的每个元素进行计算，每个核心程序以流记录作为输入并产生流记录作为输出。流程序声明流，定义核心程序之间的控制流和数据流。现有的流程序设计语言有 MIT 开发的 StreamIt[63]，斯坦福大学开发的 Brook[64]，斯坦福大学开发的 StreamC/KernelC[65]等。

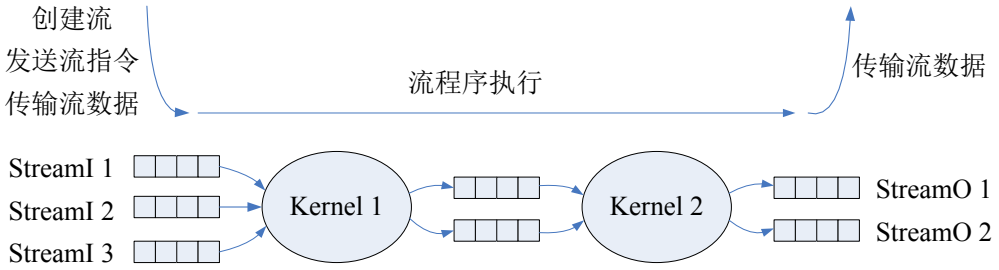


图 2-5 流编程模式

[1] 虽然该论文中将重构逻辑叫做 FPGA，但是本质上来看其实是一个粗粒度动态可重构阵列。

2.2.2 S-core 架构

一般来说，支持流编程模型的处理器就是流处理器。图 2-6 是 S-core 的架构图。整个 S-core 分为两大主要部分：存储子系统以及粗粒度可重构阵列。左侧为存储子系统，它主要包括两部分：流寄存器文件（Stream Register File, SRF）用于流数据的存储；流缓冲（Stream Buffer）用于连接流寄存器文件和重构阵列，其中每一个流缓冲都支持对一个流的操作。可重构阵列包括以下几个部分：1）运算核；目前的运算核中主要的部分为浮点的乘加器；2）互连网络；目前的互连网络为简化的星型网络（如图 2-7），简化的内容为网络只支持从左向右的数据传输。这样流数据从左侧流入阵列经过一系列的计算之后从右侧流出，并生成新的流数据。采用星型网络是因为与 2D-mesh 相比，星型网络更为灵活并且更为高效；与交叉开关相比星型网络可以扩展，并且当网络规模较大时，相比于交叉开关需要更少的互连资源[66]。3）构令存储器；存储了重构阵列和计算核之间互连的重构信息，这些重构信息设定了每一个运算核的功能以及网络如何传输数据到下一个计算节点。目前构令存储器可以存储两组构令信息，也就是 2 个核心程序，这样可以掩盖一部分传输构令流的延迟。这些构令信息是由主处理器写入的。

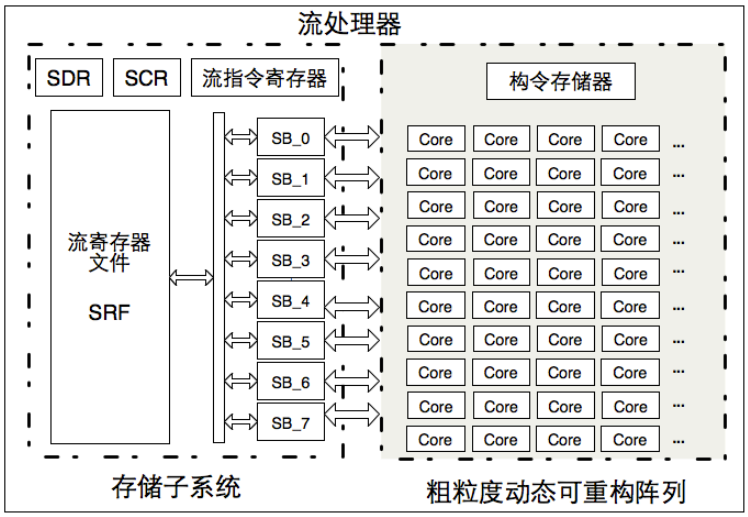


图 2-6 S-core 体系结构模块图

本文作者独立完成的存储系统设计参考了斯坦福大学 Imagine 处理器以及国防科技大学的 MASA 处理器，完成了硬件微结构的设计和实现以及软件工具包的设计。工具包包含了一系列为应用程序员提供的 API 函数，这些函数的主要工作是管理流数据，例如在片上分配 SDR 和 SRF 空间，管理核心程序的输入流和输出流等等。较为细致的设计实现内容请参见作者发表的论文《An Evaluation of the Many-core Longtium SP Computer System》以及撰写的文档《支持流模型的浮点计算平台》[1]。

[1] 可以在以下地址下载 http://pan.baidu.com/s/1pJAzk1d，S-core 在该文档撰写之后修改了互连网络，在阵列中添加了额外一组构令存储器，优化了关键路径等。该文档还详细描述了本文作者所开发的软件工具包，可以帮助应用程序员开发 S-core 上运行的程序。

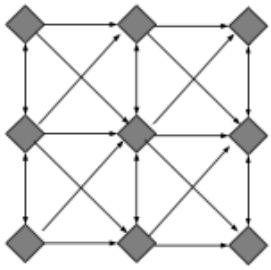


图 2-7 S-core 粗粒度可重构阵列的简化星型互连网络

2.2.3 S-core 重构除法器

可重构阵列中大部分的计算核都只包含一个融合的乘加器，这个乘加器可以支持加法、减法、乘法、乘加、乘加取反、乘减和乘减取反 7 条指令。为了计算除法，本设计使用多个计算核重构出浮点除法器。这个重构出的除法器可实现流水化操作，即每时钟周期都可以接收操作数并且在流水线饱和后每时钟周期都可以输出结果。重构除法器需要使用特殊计算核中的倒数估计值计算单元，以便将除法转变为乘法。本小节介绍除法器的算法和使用可重构阵列重构除法器的方法，本小节的目的是展示如何对粗粒度可重构阵列进行重构。

重构除法器通过使用倒数估值计算单元将除法转变为乘法，也就是将 A÷B 转变为 A×(1/B)，所以获得精确的除数倒数对于计算除法非常重要。设计使用对称表相加的方式（Symmetric Table Addition Method，STAM）来获得倒数的粗略值[67-71]，之后使用牛顿迭代（Newton-Raphson iteration）来提高倒数的精度。如果仅仅通过 STAM 就获得精确值，那么所需要的查找表将特别大，浪费资源；STAM 结合牛顿迭代是获得精确值的较为有效的方法，并且资源适中。设计使用硬件电路实现了 STAM，并将其打包成浮点倒数估值单元添加到阵列中。为提高浮点倒数值精度而设计的牛顿迭代公式见公式（2-1），其中 x_0 为 STAM 查表求和后得出的粗略值，x_1 为牛顿迭代计算后所获得的除数倒数的精确值，B 为除数。经过一次牛顿迭代可以将精度提高一倍。

$$x_1 = x_0 + x_0 \times (1.0 - x_0 \times B) \qquad (2\text{-}1)$$

了解了前面的浮点除法算法后，本文使用了四个运算核来重构出一个非阻塞的除法器，其中前三个运算核一起完成一次牛顿迭代，详细的重构结构见图 2-8。第一个运算核负责查表估算除数的粗略倒数得出 x_0；第二个运算核配置为一个乘减取反运算器求出 $1.0 - x_0 * B$；第三个运算核配置为一个乘加器求出牛顿迭代的结果 x_1；最后一个核配置成一个乘法器求出最终的除法结果。图中的蓝色虚线表示配置路径，黑色虚线表示运算器中当前运算没有使用到的部分，而黑色实线表示本运算器中当前运算实际使用部件的数据通路。

程序员可以用同样的方法将阵列重构成实现不同功能的运算单元以实现某些软件算法，例如本文同样使用 4 个运算核重构出蝶形运算器支持 FFT 运算。

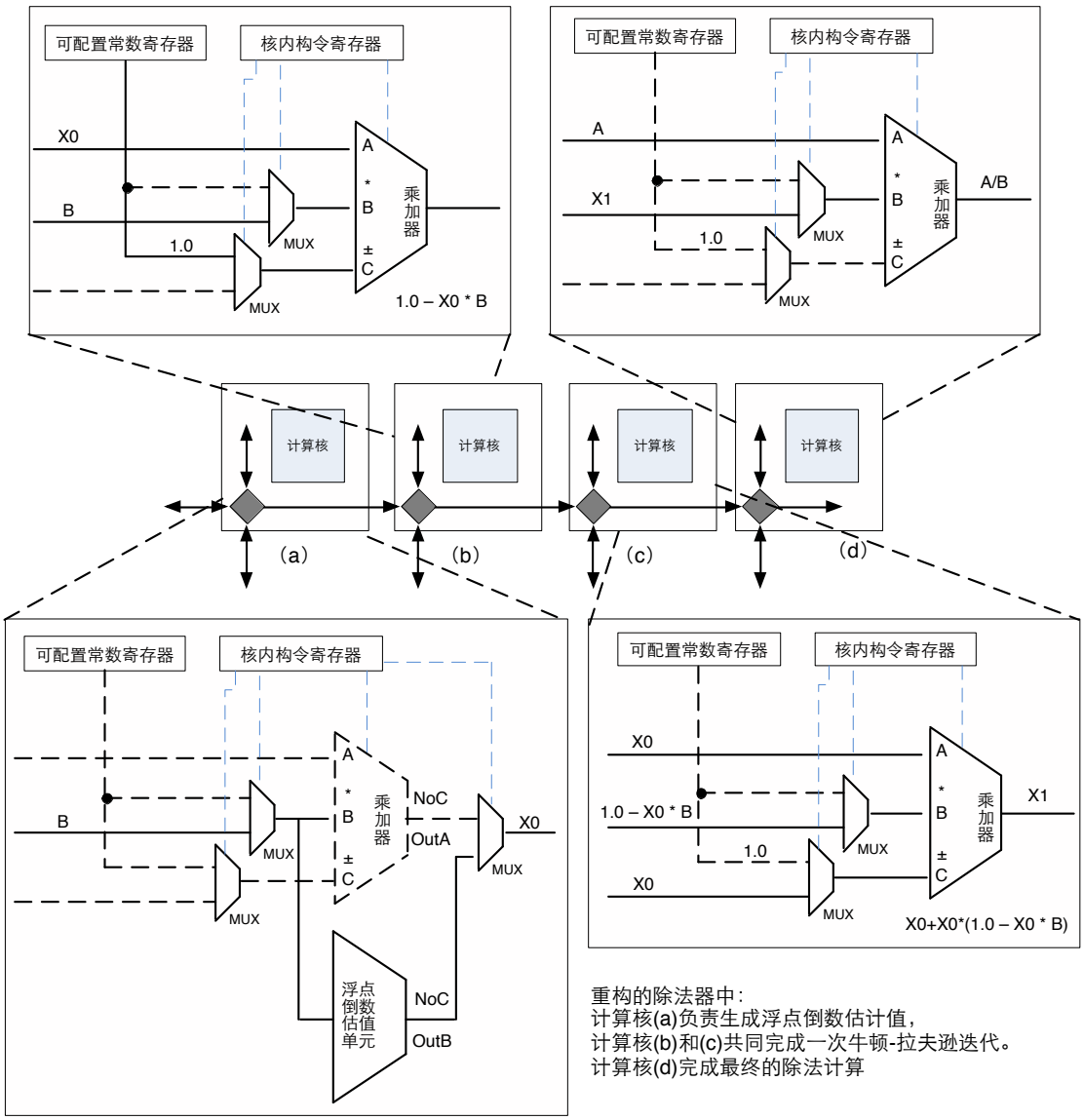


图 2-8 重构的浮点除法器结构

2.2.4 S-core 实现与评估

为了验证和评估本文的流处理器，本文使用 Altera Stratix II FPGA 实现了原型系统，系统使用 NIOS 处理器作为主处理器，并为 S-core 添加总线适配器挂载到 Avalon 总线上。由于 FPGA 资源限制，实现了 4x4 的粗粒度可重构阵列，32KB 的流寄存器文件，共占用了 69%的逻辑资源并可以正确运行在 100MHz 时钟频率下。表 2-2 列出了所有 S-core 实现部分的代码行数。图 2-9 是最终实现的原型系统。

使用本文作者编写的软件库，在 FPGA 原型系统上实现了几个典型应用程序算法，并以此来评估系统的性能。本文所实现的算法包括：1024 点复数 FFT 的 10 级蝶形运算，1000 个浮点数输入的 16 阶 FIR 数字滤波，2048 点的浮点向量平方和计算和 2048 点浮点除法。所有计算结果和 C++程序在 Intel 平台上的计算结果进行对比，全部运行正确（由于 Intel 处理器浮点栈为 80 位宽，而 S-core 是 32 位宽，所以结果略有差别，但是误差在精度所接受范围内）。评估性能的流处理器中可重构阵列使用 16 个运算核，被比

较的单核是兼容 PowerPC 750 的龙腾 R 处理器，运算核中的运算部件就取自该处理器。流处理器上程序的执行时间通过读取特殊的硬件计数器获得，在龙腾 R 处理器中也添加实现了类似的计数器。表 2-3 和图 2-10 分别是执行拍数和加速比。

表 2-2 S-core 代码量统计

模块	行数	字数	字符数	字数占比
Top Level	1089	2741	36562	7.54%
Stream Buffer	991	2498	34995	6.87%
SRF	4571	9919	153888	27.29%
Computing core	5906	13318	131121	36.63%
Star-Mesh	2086	4429	45731	12.18%
Software API	1485	3450	33900	9.49%
Total	16128	36355	436197	100%

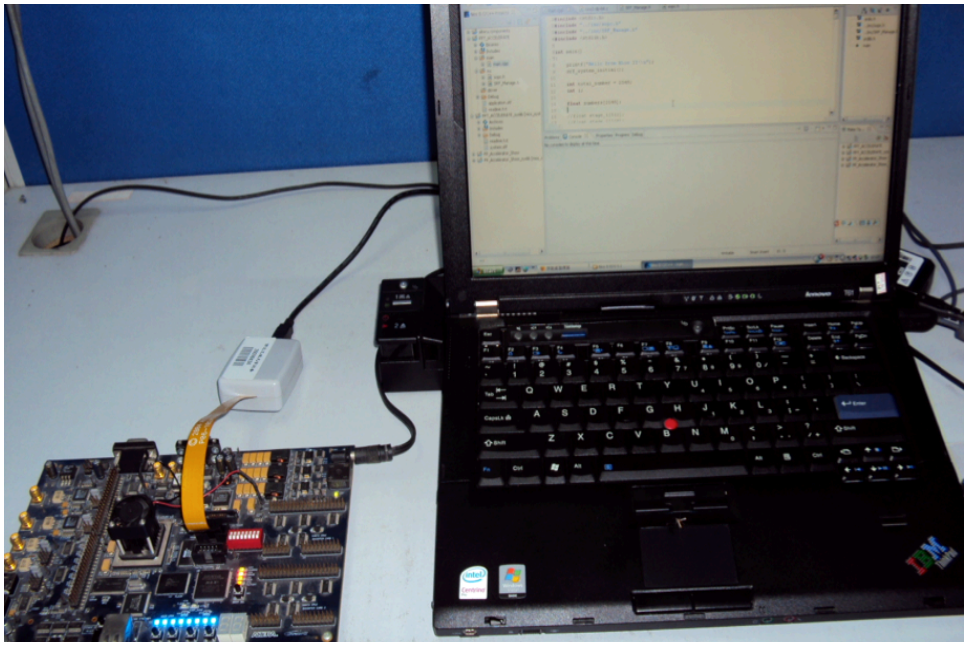


图 2-9 FPGA 原型系统

表 2-3 龙腾 R 和 16 核下的执行拍数

	FFT（R2 n=1024）	FIR(R16 n=1000)	向量平方和（n=2048）	浮点除法（n=2014）
单核	41040	29465	2072	8182
16 核	3470	1370	326	552
加速比	11.83	14.21	6.36	14.0

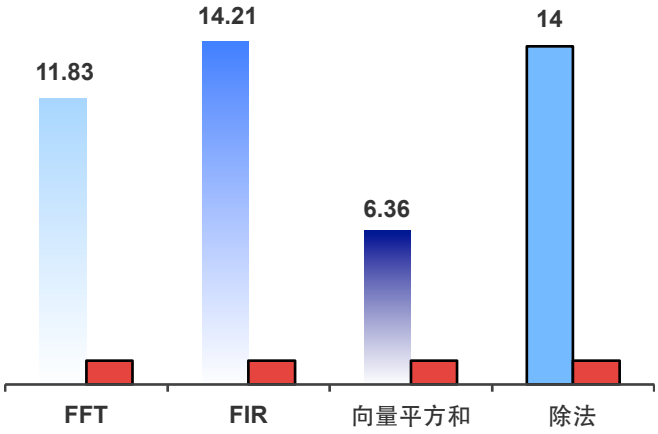


图 2-10 应用程序计算加速比

2.3 移动应用处理器 GreenDroid

移动应用处理器不但受到芯片散热和稳定性方面的制约，更受到供电电池容量的制约，因此受到使用墙的影响更为突出。公式（2-2）是移动设备受供电电池影响的功耗预算计算公式，通常移动应用处理器的功耗预算仅仅只有几百毫瓦。另一方面，由于移动设备已经取代传统桌面电脑和笔记本电脑成为主流的计算平台，所以研究解决移动应用处理器的使用墙问题将是解决暗硅现象的有效切入点。图 2-11 是预测机构 Gartner 对近期设备出货量的预测，可见未来移动设备出货量将是传统电脑的 7 倍以上。图 2-12 是 Gartner 关于安装不同操作系统的设备出货量预测，从图中可以看出安卓(Android)设备占据出货量的一半以上，并且所占比例还在不断提升。此外由于安卓的开源策略使得大家可以十分容易的获得该系统并对其进行分析。因此团队决定首先面向安卓移动操作系统提出了一款集成大量专用协处理器的移动应用处理器 GreenDroid[1]。

$$Power\_Budget_{battery} = \left(\frac{battery\_capacity}{\#hrs\_active\_use\_between\_recharges}\right) \tag{2-2}$$

Worldwide Device Shipments by Segment (Thousands of Units)

Device Type	**2013**	**2014**	**2015**
Traditonal PCs (Desk-Based and Notebook)	296,131	276,221	261,657
Ultramobiles, Premium	21,517	32,251	55,032
PC Market Total	**317,648**	**308,472**	**316,689**
Tablets	206,807	256,308	320,964
Mobile Phones	1,806,964	1,862,766	1,946,456
Other Ultramobiles (Hybrid and Clamshell)	2,981	5,381	7,645
Total	**2,334,400**	**2,432,927**	**2,591,753**

Source: Gartner (June 2014)

图 2-11 Gartner 对传统 PC 和移动设备出货量预测

[1] 本小节更多信息请参见 www.greendroid.org

Worldwide Device Shipments by Operating System (Thousands of Units)

Operating System	2013	2014	2015
Android	898,944	1,168,282	1,370,893
Windows	326,060	333,419	373,694
iOS/Mac OS	236,200	271,115	301,349
Others	873,195	660,112	545,817
Total	**2,334,400**	**2,432,927**	**2,591,753**

Shipments include mobile phones, ultramobiles (including tablets) and PCs

Source: Gartner (June 2014)

图 2-12 Gartner 对安装不同操作系统的设备出货量预测

2.3.1 安卓架构分析

为了探索安卓是否适合专用化，本文对安卓软件栈架构进行了简单的分析。图 2-13 展示了安卓系统的软件栈。其中安卓系统的核心是一组用 C 和 C++编写的原生软件库（Native Libraries），这些软件库为系统提供了大部分所需的服务，例如压缩、窗口合成、2D 和 3D 图像等功能。这一层软件还包括 Dalvik 虚拟机(DVM)1。Dalvik 虚拟机执行用户应用程序的 Java 代码并编译到底层的字节码，虚拟机提供的 Java Native Interface(JNI)使得 Java 程序可以通过这个接口调用原生库。

安卓应用程序中的“热代码”通常是调用原生库实现的，而“冷代码”就在 Dalvik 虚拟机上运行。这也使得 Dalvik 虚拟机成为了热代码。通过这种运行方式，应用处理器上大部分运行的热代码都是安卓软件栈所提供的。

安卓这样的内部架构就决定了系统运行高度依赖于一组共享的软件库，同时对于移动平台还有一组较为常用的典型应用，例如浏览器、视频软件、音乐播放器和邮件应用等等。针对这些共享库和常用的应用定制专用 C-core 并集成到 GreenDroid 中就可以显著降低系统能量消耗。定制的专用协处理器分为 2 种：1）针对原生库和 Dalvik 虚拟机中的关键代码进行定制，这种专用协处理器可以降低整个系统中所有应用程序运行时的能量消耗，本文的实验表明采用这种方式可以潜在地使 C-core 平均覆盖 72%的应用程序执行。2）在芯片面积允许的情况下，可以针对安卓用户使用的常用软件进行进一步专用化。根据芯片面积情况，这种方式可以潜在地使 C-core 覆盖应用程序执行的 80%甚至可以到 95%。尽管这种 C-core 对更新版本后的安卓系统可能无法使用，但是移动终端设备具有较高的替换率，通常 2 年之内就会替换新的设备，这就使得架构师可以持续开发新的 C-core 来覆盖新版本的操作系统，并将老的 C-core 从系统中移除。

以上分析表明集成 C-core 可以覆盖基于安卓的移动设备上运行的应用程序中大部

[1] Dalvik 虚拟机生成的字节码还要再次被编译转化为机器码，所以一直被认为是拖慢安卓系统速度的根源，在 2014 年 6 月的谷歌 I/O 大会上，Dalvik 被 ART 所取代，但是本文撰写时没有对新的 Android 架构进行详细分析，当时初步观察原生库还是存在的。

分的代码。此外，随着晶体管工艺的不断进步，芯片上的晶体管资源将越来越多，架构师就可以利用更多的资源来实现 C-core，进一步的覆盖更多的软件代码。基于此提出了 GreenDroid 移动应用处理器，处理器的架构见图 2-2(a)。

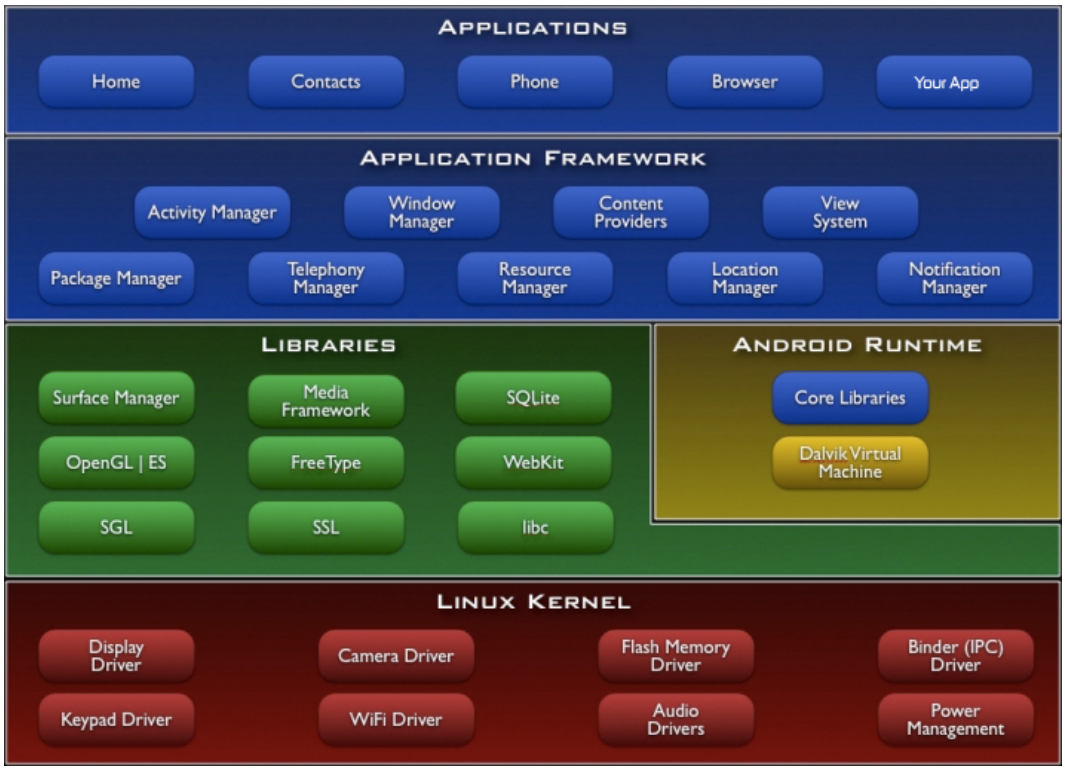


图 2-13 安卓操作系统软件栈

2.3.2 GreenDroid 能耗分析

本文对 GreenDroid 其中的一个瓦片进行了重点的评估，该瓦片布局布线后的版图见图 2-14。每一个瓦片都有包含通用处理器核、数据 Cache 和指令 Cache 以及片上网络接口。此外这个瓦片还包含 9 个 C-core。这些 C-core 都是针对安卓的 2D 图像库（7 个，针对的 libskia）、JPEG 解压缩(1 个)和快速傅里叶变换(1 个)功能定制的。使用 TSMC 45nm 工艺，这些 C-core 使用了 0.58mm^2 的芯片面积，是这个瓦片面积的 58%。

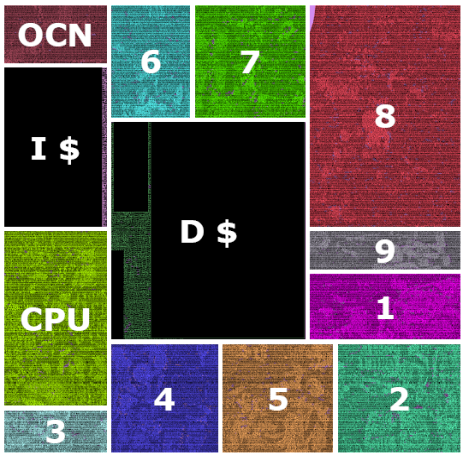


图 2-14 GreenDroid 中一个瓦片的版图

图 2-15 展示了 GreenDroid 集成 C-core 后对能量消耗的优化效果。左边是应用在主处理器上运行时能量消耗的平均值，右边为在 C-core 上执行部分的能量消耗平均值。节约的能量主要来自于两方面：1）程序在 C-core 上执行不需要取指、译码以及寄存器文件等相关逻辑，这样可以使能量消耗减少 56%；2）另外 35%的能量节约来自于定制的 C-core 数据通路。程序在 C-core 上执行每条指令的平均能量消耗从 91pJ 变为 8pJ。

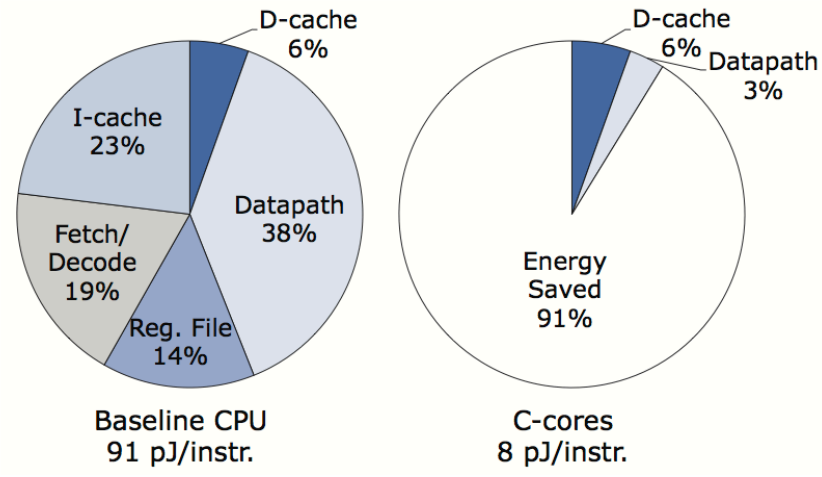


图 2-15 GreenDroid 系统集成 C-core 后对能量消耗的优化

为了提升 C-core 对整个系统的优化效果，架构师就要不断提高程序在 C-core 上运行的覆盖率。覆盖率的提高意味着将要集成更多的 C-core，这就带来了芯片面积的开销。下一小节通过一个实际例子来评估面积开销的大小。

2.4 硅实现的浏览器 SiChrome

浏览器是移动设备最重要的应用，根据尼尔森（Nielsen）2011 年移动设备运行的应用分析，安卓智能手机用户花费了 31%的时间使用浏览器，并且对于功能机来说浏览器甚至是唯一的应用程序。因此浏览器不但是移动设备上运行时间较长的应用消耗较多的能量，而且由于浏览器属于控制复杂、并行性较差的应用，较为适合 C-core 特点。

2.4.1 安卓浏览器分析

为了理解安卓浏览器的工作情况，本文使用程序剖析器(profiler)来分析程序运行。本文使用基于 QEMU 的安卓模拟器来搭建完整的安卓系统，包括安卓共享库、Dalvik 虚拟机以及 Linux 内核。使用 Trace 的方法记录所有 CPU 上执行的线程中的每一条指令。这些指令包括了来自于原生库、Dalvik 虚拟机和 Linux 内核中的代码。整个应用程序分析的过程完全不需要修改应用程序的代码，仅仅需要修改模拟器并添加一些用于统计的代码。

本文使用修改过的安卓模拟器来重点分析了浏览器访问网站时的执行情况，访问的网站包括将近 20 个常见的网站，例如 Google，gmail，Google News，Google Finance，Amazon, Google Docs 等等。本文试图分析出以下信息：1）静态指令数量，也就是浏览

器执行所使用的代码段一共有多少条静态的指令，这个指令的数量决定了最终生成的 C-core 的面积；2）线程关系，影响了 C-core 的分布；3）代码共享和并发执行，影响了 C-core 的复制；4）函数调用图、静态函数数量以及预期执行覆盖率，这些因素决定了 C-core 的数量。

图 2-16 是按照库分类的应用运行时间占比图。一半以上的运行时间都被做渲染用的 libwebcore 库消耗、其余图形库 libskia 占用了 10%的运行时间、libc 库占用 9%、Dalvik 虚拟机库 libdvm 占用了 8%、linux 内核占用 7%等等。从图中可以看出浏览器运行时，大部分的运行时间都是被各种各样的原生库、虚拟机和内核代码占用。架构师只要将这部分代码硬件化，就可以使得 C-core 覆盖程序大部分的执行，这样就可以大幅度的提高系统的能量效率。换句话说，大量集成 C-core 的这种方式可以有效覆盖浏览器执行，这也证明了 CoDA 架构理论上是适合暗硅时代的移动平台处理器。

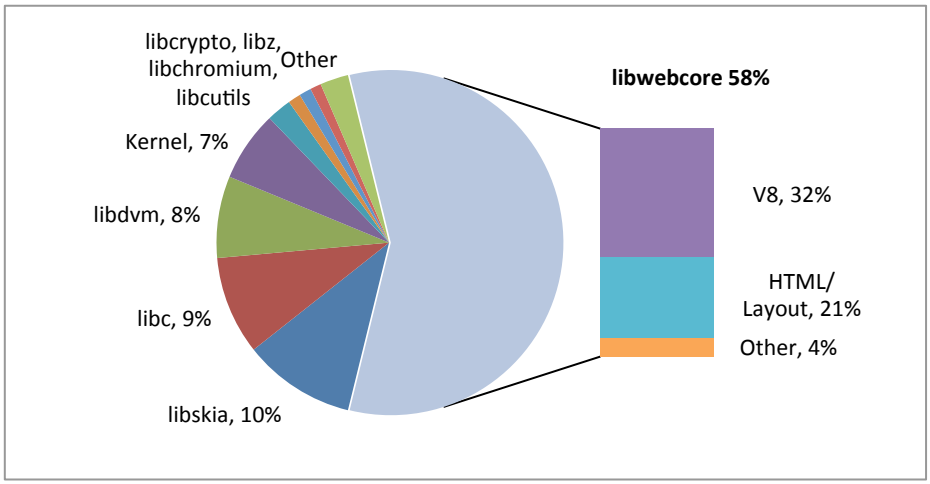


图 2-16 按库分类的执行时间

图 2-17 是静态指令数量与动态执行时代码覆盖率之间的关系。横坐标是静态指令的数量，纵坐标是动态执行时对应静态指令可以覆盖多少执行指令数。本文先把静态指令段按照动态执行时被执行的次数进行排序，然后从高到低逐次统计。由红色曲线可以看出，刚开始由于添加的静态指令被重复执行次数较多，所以覆盖率增长较快；之后添加的静态指令被重复执行次数较低，所以覆盖率增长缓慢。图中箭头所指的点为覆盖 90%动态执行时所需要硬件化的静态指令数，大约为 84000 条左右。通过计算 2.1 小节中 C-core 平均实现一条特定指令所需的硬件资源，来评估硬件化这 8 万多条指令所需要的资源。最后评估出在 22nm 工艺下大概需要 7mm^2，考虑到目前商用的处理器大约具有 100mm^2 左右的 Die 面积，以及使用未来新工艺之后晶体管密度增加，芯片资源应该是可以满足硬件化的需求的。此外，由于硬件化所针对的都是共享的原生库、虚拟机和系统内核，这些硬件化后的资源也是可以被其他应用程序共享使用的。

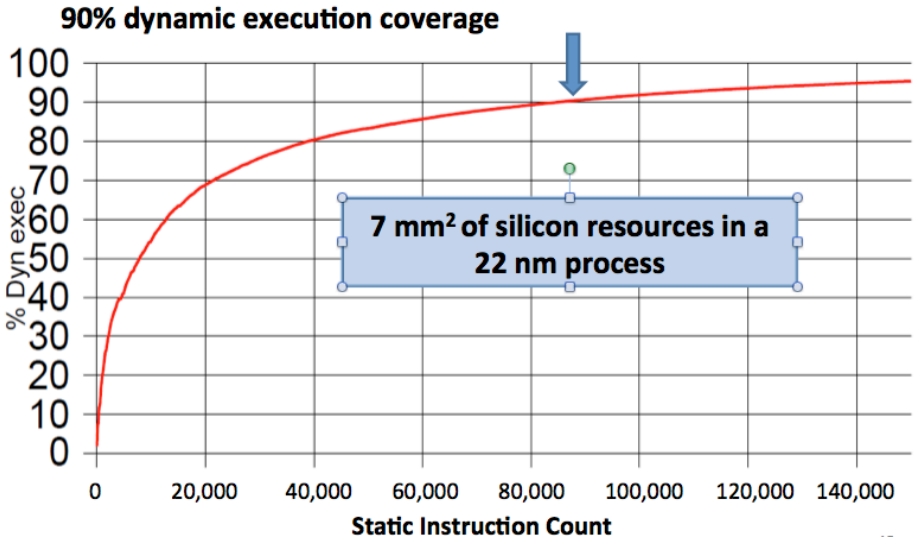


图 2-17 静态指令数量与动态执行覆盖率关系图

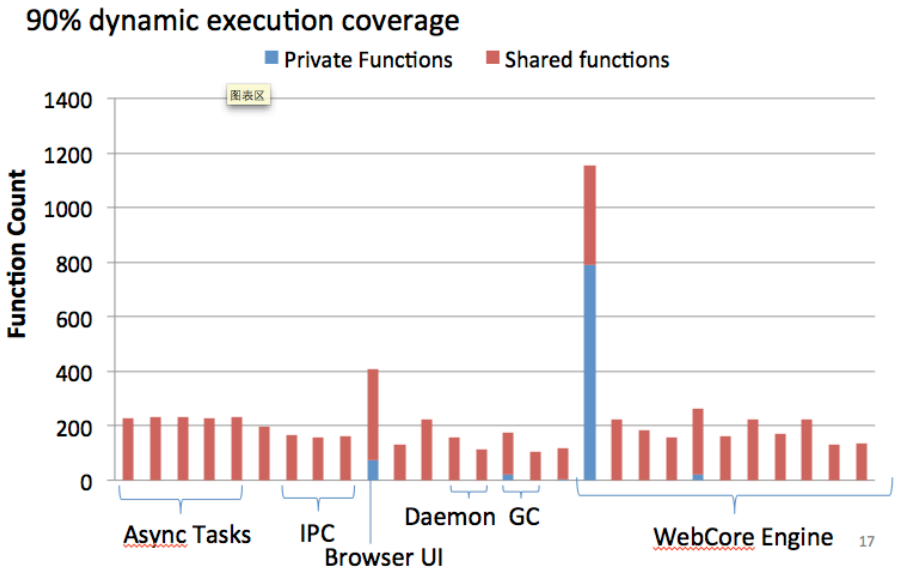


图 2-18 各个执行部分私有函数与共享函数比例关系

此外，本文对静态函数与执行覆盖率也进行了统计，当覆盖 90%的代码执行时，需要有大约 1200 个静态函数；而覆盖 80%代码时需要大约 625 个静态函数。这些静态函数数量决定了生成的 C-core 的数量。

图 2-18 展示了执行各个功能部分中私有函数与共享函数之间的关系和数量，从中可以看出程序大部分执行需要使用的都是共享函数。设计师仅仅需要将这些共享函数硬件化就可以获得较高的代码覆盖率。

2.4.2 CoDA 适用性分析

本章首先通过 GreenDroid 分析了安卓移动平台的架构，发现这个平台大部分的应用运行都是基于共享原生库和虚拟机的。只要将这部分代码硬件化，就可以潜在地使得应用程序在专用协处理器上运行时间占到总时间的 72%。另外，还可以针对安卓系统流行

的 APP 设计更多的仅仅适用于这些 APP 的专用协处理器，以便覆盖更多的应用程序执行部分。这样可以使专用协处理器潜在地覆盖 80%甚至 95%的应用程序执行。

作为对上述理论分析的补充，本节以浏览器为实例重点分析了浏览器的执行情况，以及硬件化浏览器所需硬件资源。通过本小节的分析，知道将应用程序硬件化并获得足够高的代码覆盖率在硬件资源的角度来说是较为可行的。系统中集成 1000 个左右的 C-core 并且占用芯片资源的一小部分就可以覆盖重要应用浏览器的大部分代码执行，并且这些硬件化的代码大部分都是共享代码，也是其他应用需要使用的。

本章从两个层次分析了 CoDA 对应用的适用性。第一，目前软件编程大多基于库函数和其他一些共享代码，将这部分代码硬件化并集成到 CoDA 架构就可以覆盖大部分软件代码的执行，所以 CoDA 架构适合这种软件架构。第二，通过评估硬件化浏览器所需的芯片资源，发现使用可接受的硅面积就可以覆盖应用执行。这就证明了 CoDA 适用于应用，并可以适应暗硅时代芯片资源特点。但是集成这些 C-core 也会带来大量的集成开销，例如互连、存储以及漏电功耗等等。这些开销对系统整体能量效率的影响将是本文之后探索的重点。

2.5　小结

本小节首先介绍了全文的基础，函数粒度专用协处理器 C-core 的自动生成工具链、硬件结构、集成方法、程序跳转执行以及 C-core 对能量效率的影响。评估发现对于单个应用，在通用架构中集成 C-core 就可以节约测试程序 70%的能量消耗。其次，针对某些特殊的、适合流计算的、具有生产者消费者局部性的功能函数提出了 S-core 架构。再次，针对当前移动系统迅猛发展以及电池供电设备的限制，选择安卓系统进行了分析。分析发现大部分应用程序是基于原生库和虚拟机的，只要把这部分软件代码硬件化就可以使得专用协处理器覆盖应用较大部分。之后，本章以安卓浏览器作为典型应用进行了重点分析，发现 CoDA 确实适用于安卓系统并且可以适用于暗硅时代的移动平台处理器。本文还分析了 SiChrome 所需硬件资源，本文认为未来芯片片上资源是可以满足所提出的架构的，这也是后续章节研究 CoDA 的重要基础。

本章第一小节内容来源于文献[5, 13, 19]，以及本文作者参与的对 C-core 结构进行优化的后续工作，包括进一步修改了 C-core 中的多时钟周期路径、运算器进行了融合、满足交换律的运算通路进行了更好的路径平衡等等。第二小节来自于作者参与的 863 项目以及发表的论文“An Evaluation of the Many-core Longtium SP Computer System”。第三小节和第四小节发表于论文“GreenDroid: An architecture for dark silicon age”和“SiChrome：A Silicon Browser”。

3 CoDA 可扩展性和建模

在暗硅时代，芯片架构师遇到越来越严重的使用墙问题，使得向通用处理器架构中集成高能量效率的专用协处理器越来越受到关注。使用专用协处理器来适应暗硅时代的关键是：即使对于大规模、多样地负载来说，也要能有效降低负载执行过程的能量消耗，这就需要在架构中利用暗硅来集成大量的专用协处理器。前文的工作证明了 CoDA 对程序的适用性，本文后续章节主要来研究如何有效地在 CoDA 架构中集成成百上千的专用协处理器以及集成这些专用处理器之后系统的能量效率和扩展性问题。

此外 CoDA 架构也会带来关于专用协处理器对应用执行覆盖率以及使用模型方面的问题。这些都需要对 CoDA 架构进行较为仔细的研究。但是 CoDA 结构复杂，可以选取的设计参数较多故而存在大量 CoDA 的可行设计和实现，RTL 实现、综合并仿真所有这些 CoDA 设计实现是不可能的。所以本章提出了一种分析 CoDA 架构性能、面积和能量效率的分析评价模型，并描述了这个模型的参数空间，下一章将使用这个分析模型来探索 CoDA 架构。

本章的主要内容如下：第一小节描述了 CoDA 多维可扩展架构、能耗管理策略和执行策略；第二小节描述 CoDA 代码覆盖率和目标应用负载；第三小节描述本文提出的 CoDA 分析模型；第四小节描述 CoDA 设计的参数空间。第五小节对本章进行了总结。

3.1 CoDA 架构

针对大规模应用负载所设计的具有广泛适应性的 CoDA 架构中将包含成百上千的专用协处理器，要求本文提出一种多维度可扩展的体系架构。本小结将描述以下内容：1）为了使 CoDA 设计能够覆盖较大规模的应用以及更好的适应应用的不同特点，CoDA 设计所采用的多维可扩展架构；2）CoDA 架构的能耗管理策略，并描述了每一个核都是如何被点亮的；3）描述了本文 CoDA 架构中程序执行调度策略。

3.1.1 多维可扩展 CoDA 架构

大规模 CoDA 架构所支持的应用程序具有各种不同的特点。有的程序需要使用更多数量的专用协处理器，而有的程序仅仅需要使用几个专用协处理器；有的专用协处理器较大，有的协处理器较小，不同协处理器之间可能具有几倍的面积差距。专用协处理器的分布也会对 CoDA 架构的性能、能量消耗产生影响。此外，主流的瓦片化多核架构需要瓦片的面积大致相等。在这些需求约束下，CoDA 架构需要在多个不同的维度上提供可扩展性，才能灵活的适用于不同应用程序集，并提高系统的能量效率。图 3-1（a）展示了本文研究的 CoDA 设计的高层次架构，下文分层次介绍这个架构的可扩展性。

在瓦片扩展维度（Tile-core Scaling）上，本文的 CoDA 设计可以由不同数量的瓦片构成。瓦片可以是不同处理器核集合，也可以是二级 Cache。如图 3-1（a）所示，CoDA 架构中这些不同数量的瓦片还被组织成多个电压域（Voltage Domain）。每一个电压域都

包含一个或多个运算瓦片和一个共享的二级 Cache 瓦片。这些瓦片之间的通讯使用点到点的、虫洞路由的 2D-mesh 物理互连网络，而不是虚拟通道。使用 2D-mesh 作为片上网络是因为这种网络具有较好的众核扩展能力，这使得 CoDA 可以集成无限的瓦片进而集成无限多的专用协处理器，大大扩展了探索空间。

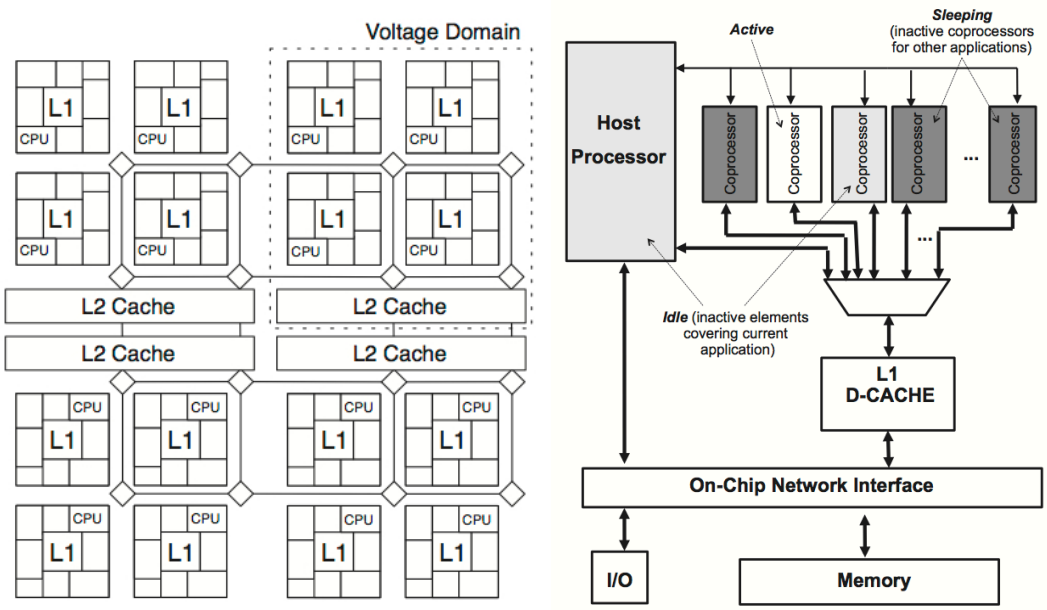


图 3-1 (a) CoDA 原型 (b) 紧耦合的专用协处理器集成

在瓦片内部扩展维度（Intra-Tile Scaling）上，CoDA 中每一个运算瓦片都可以包含数量不同的专用协处理器，也就是瓦片内专用协处理器的数量可扩展。为了使编程模型保持不变，设计不但需要使专用协处理器兼容函数调用 ABI 还要让专用协处理器与通用处理器共享 Cache 来避免不必要的数据调度。这样在瓦片内部主处理器和专用处理器通过一组树形网络连接 Cache，并在同一时刻仅仅允许一个处理器访问 Cache，使得瓦片内形成了局部多核架构。集成的专用协处理器越多，Cache 访问路径上的多选器也就越大、延迟越长，这约束了瓦片内可以集成专用协处理器的数量上限。如果集成的专用处理器数量太多，无法在一个瓦片内集成，那么需要在高层次增加瓦片的数量。图 3-1（b）展示了一个瓦片内部各个部件的互连情况。如图所示，运算瓦片中还有一致的 L1 指令和数据 Cache，一个动态互连网络路由。主处理器与专用协处理器的连接也使用树形网络。

在专用协处理器扩展维度（Intra-Coprocessor Scaling）上，每一个被集成到 CoDA 架构中的专用协处理器都可以是异构的，并且这些异构的专用协处理器的大小和支持的功能都不相同。有的仅仅支持几个程序基本块，有的可以支持几百个程序基本块。此外，架构师还可以根据需要通过编译优化和手工代码优化来指导专用协处理器的设计实现。例如，通过内联函数（inline）可以使得专用协处理器变大，并减少程序在主处理器和专用协处理器之间的跳转次数。

运行在某一瓦片上的线程可以使用该瓦片内任何一个专用协处理器；如果该线程需

要使用该瓦片内没有的其他专用协处理器，线程需要迁移到包含该特定专用协处理器的瓦片上执行。本文不对瓦片内多个专用协处理器同时执行以支持多线程的执行模式进行讨论，但是该种情况可以通过激活多个瓦片，并使得每一个瓦片执行一个线程的方式进行覆盖。

在 CoDA 中，某一个瓦片内的处理器核访问该瓦片内一级 Cache 的延迟与该瓦片中集成的专用协处理器数量和某一个指定的专用协处理器与一级 Cache 之间距离有关，而 L1 Cache 的访问延迟是影响处理器性能的关键因素。较大面积的瓦片由于集成更多的专用协处理器，会导致 L1 Cache 访问延迟加大，这将限制处理器的性能。另外在实际应用中，瓦片内的所有专用协处理器或者不同瓦片对存储器结构和存储器延迟的敏感程度都不相同。在 CoDA 设计中，由于瓦片内可能集成较多的专用协处理器，本文通过 Profile 的方式来提取每个专用协处理器对存储带宽和延迟敏感程度的需求，然后将需求较高的专用协处理器放在距离一级数据 Cache 较近地方，而需求较低的放在距离一级数据 Cache 较远的地方。该方法不但可以减少线的总长度，降低多路选择器的功耗并且由于线延时的缩短减少对性能的影响。这种方法与采用对称地多路选择器，并提供相同的存储器延时的解决方案具有本质的区别。

CoDA 架构使得代码几乎不在通用主处理器上运行，并且系统中主处理器的数量等于瓦片的数量。因此主处理器的性能相对于主处理器的简单性来说，显得不那么重要. 因此瓦片内的主处理器最好选用高能量效率的顺序执行处理器，并且支持快速从深度休眠唤醒。本文设计中所采用的主处理器是兼容 MIPS 的 MIT Raw 处理器[72]。通用主处理器通过树形互连网络可以访问专用协处理器内部状态寄存器，并通过读写这些状态寄存器来控制协处理器的运行。

3.1.2 CoDA 能耗管理策略

本文探索的 CoDA 架构支持三种功耗和能量管理方式，分别是多电压域、门控电源和门控时钟。本小节将描述这些功耗和能耗管理策略的具体应用情况，以便读者更好地理解 CoDA 设计与传统架构的核心差别并更好的理解为什么 CoDA 可以适应暗硅时代。

通过在 CoDA 架构中引入多个电压域（图 3-1(a)中虚线部分）将整个瓦片阵列划分为多个小阵列。每一个电压域都有自己的电源网络（power rail），并使用芯片外的电压控制器（voltage regulator）来控制这些电源网络。通过采用这种设计使得系统可以完全关断不需要使用的电压域，这样做的代价是线程将不可以再利用其它关断的电压域中的二级 Cache 资源。为了探索 CoDA 架构的设计空间，本文假定冲刷 Cache 和使电压域上电或掉电需要几百微秒的时间，改变激活的电压域仅仅发生在操作系统对应用进行调度时。

架构所使用的门控电源具有深度睡眠的能力并且上电和断电所需时间也和操作系统调度时间类似。CoDA 架构以应用程序为粒度配置需要上电的专用协处理器。为了高

效的管理没上电的闲置资源，系统使它们在默认状态下处于深度的关断电源状态，操作系统根据应用以及并发性来配置共享资源。因为应用程序需要使用哪些专用协处理器具有高度的可预测性并且这些专用协处理器就是根据应用而生成的，所以在程序执行过程中需要唤醒专用协处理器的情况不经常发生。大部分情况仅仅需要操作系统在调度程序执行时进行配置。操作系统的调度时间与现在常用的门控电源技术[73]以及更为激进的门控电源技术[74-76]具有类似的时间。

CoDA 设计还考虑了门控时钟的使用。因为一个应用程序可能会使用多个专用协处理器，而且使用可能是串行的。这样处于闲置状态的专用协处理器，如果和当前运行或正在调度的应用有关则使它处于门控时钟关断状态；如果无关则处于关断电源状态。这种形式和最新的基于 big-LITTLE 架构的移动应用处理器类似[17]。通过这样的方式，可以在较粗的粒度上节约大量能量消耗。

此外，本文的 CoDA 架构在同一时刻，每一个瓦片内部仅有一个执行单元处于激活的状态，该执行单元或者是主处理器或者是某一个专用协处理器。

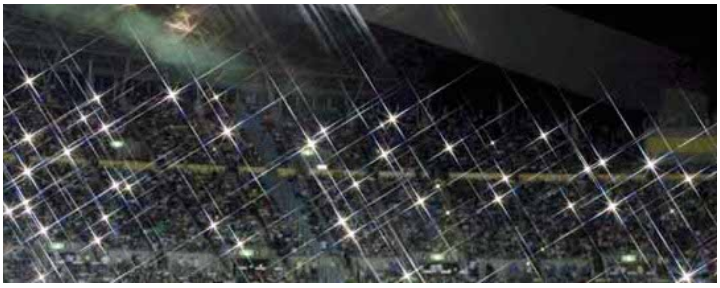


图 3-2 体育场相机闪光灯

为了更为清晰的对 CoDA 能耗管理策略进行描述，更清晰地表达 CoDA 能耗管理策略与之前传统多核/众核架构的区别，以及更清晰地统一描述功耗管理、CoDA 架构和程序执行之间的联系，本小节以体育场相机闪光灯为例进行类比说明。

图 3-2 是足球世界杯比赛现场观众席的闪光灯。如果把闪光灯闪光瞬间的光亮对应为 CoDA 中的处理器核上电后在执行程序，那么图中黑暗的地方就相当于非激活区域，也就是 CoDA 中某些处理器核所在电压域没有供电。闪光灯光亮周围还有一些灰色的区域，这部分区域对应于 CoDA 架构中电压域上电但是被门控电源关断电源或者被门控时钟切断时钟的区域。

CoDA 上运行程序的过程与体育场闪光灯光亮出现过程也有相似之处。在球员入场之前，观众席一片黑暗，此时对应于 CoDA 架构刚上电时默认情况下都处于暗硅状态。球员入场之后，一部分感兴趣的人将相机打开准备拍照，此时对应于操作系统调度程序执行并将程序执行所需的专用协处理器核所在电压域上电。由于程序的执行过程所需使用的专用协处理器大部分可以预知，所以此时操作系统将程序要使用的处理器核置于关断时钟的状态，这类似于体育场的某些观众此时已经半按了快门。上电电压域中程序不使用的处理器核需要处于门控电源的门控状态下。CoDA 架构中的程序在不同专用协处

理器间跳转执行，执行到的处理器核就会打开门控时钟进行正常计算，对应于完全按下快门闪光灯开启。程序跳转到下一个处理器核执行后，当前的处理器核再次回到门控时钟的状态，直到下一个程序被操作系统调度。这个过程类似体育场观众席上闪光灯不停的熠熠闪烁。

3.1.3 CoDA 执行策略

由于本文的 CoDA 设计所采用的专用协处理器都兼容程序调用的二进制接口 ABI 并与主处理器共享 Cache，所以无需对高层次编程语言编写的程序源代码进行修改。

在 CoDA 架构中，程序在专用协处理器和通用处理器之间跳转执行。为了在程序中插入这些跳转，CoDA 的编译器需要修改专用协处理器支持的函数调用接口 ABI，修改的内容是添加一个新的分支语句。这个分支语句检查专用协处理器是否存在并空闲。如果协处理器存在并空闲，代码可以跳转到协处理器执行；而当协处理器不存在或者忙的情况下，程序可以继续在主处理器上执行未经修改的软件代码。从其他程序部分的角度来看，这些修改后的函数调用接口部分和原来的调用是一致的。从 CoDA 设计的视角来说这种一致性是必要的也是设计的基础。这种一致性也使得 CoDA 仅仅是编译器可见的，而对于程序员来说是透明的。CoDA 中动态检查硬件的机制也用来解决多个线程对特定专用协处理器的竞争、有缺陷的部件以及兼容老代码和老的硬件。

当多个程序或者线程在 CoDA 架构中并发执行时，他们可能会竞争特定的专用协处理器。为了解决这个问题，新的函数接口可以检查所需的协处理器是否可以获得并且可以在线程迁移过来之前预定这个协处理器。如果协处理器不可获得，新调用接口可以调用原来在通用处理器上执行的代码继续在通用处理器上执行。如果 CoDA 架构中使用的协处理器支持上下文切换（像 Conservation cores [5, 13]），那么 CoDA 的编译器需要生成额外的代码使得已经开始执行的专用协处理器可以在软件需要的时候结束执行。

本研究假设所有竞争专用协处理器失败的程序或线程都回到主处理器上继续执行，直到下一次调用专用协处理器。

3.2 CoDA 协处理器和负载

本节首先介绍模型所关注的专用协处理器 C-core，这部分内容是对第二章简介的补充。此外本文的模型也适用于所有其他类型的专用协处理器，只要他们可以和主处理器紧耦合集成并且共享高速缓存；之后，本小结还描述了为评估 CoDA 架构所使用的应用程序负载

3.2.1 协处理器与代码覆盖率

可供设计师选择添加进 CoDA 架构中的协处理器种类繁多，不同种类的专用协处理器在性能、效率和潜在代码覆盖率等方面提供了不同的权衡。在创建新的专用协处理器时，大量的工程设计工作成为主要的工作瓶颈，所以 CoDA 采用了可以自动生成的专用

协处理器。专用协处理器自动生成技术大大减少了工程设计工作，使得架构师可以较为容易的向可扩展系统中集成大量的专用协处理器以覆盖大量不同种类的应用。为了观察CoDA 架构在扩展性方面的局限，本文非常保守的限制系统仅仅使用可以从非规则代码中自动生成的专用协处理器。因此，本文研究的重点落在了自动生成的专用协处理器上，这些处理器在不使用复杂的指针分析和代码转化的情况下，就可以获得较好的能量延时积优化。此外，本文所讨论的 CoDA 架构适用于很多其他类型的协处理器，只要他们可以如图 3-1（b）所示紧耦合到主处理器并且彼此共享存储器。

第二章介绍的 C-core [5]是一种满足上述要求的可以自动生成的高能量效率专用协处理器。因为可以为几乎任何功能的代码自动生成 C-core，所以架构师可以简单地生成大量的 C-core 来提高系统中专用协处理器的代码覆盖率。因此本章的研究中使用 C-core 作为专用协处理器的模型。文献[3-6, 13]表明在运行相同功能的软件代码时与顺序执行的通用处理器相比，C-core 可以降低 10 倍的能量消耗（对于非存储器操作降低 30 多倍能耗[3, 4, 6]，整个应用的能耗降低 10 倍）并且获得 23 倍以上的能量延时积优化。C-core 设计的出发点侧重于降低程序运行所消耗的能量而不是直接提高应用程序运行的速度。然而，由于 C-core 执行每条指令时功耗降低，那么在相同的芯片功耗预算限制下，CoDA 架构将支持更多的程序并发执行。

每一个单独的 C-core 都可以覆盖目标程序中的某一指定部分。使用工具链可以从应用程序的源代码中自动生成 C-core。通用处理器上运行应用程序中没有被协处理器覆盖的冷代码，例如系统启动代码、关机执行的代码以及其他一些不经常运行的代码。在程序执行时，线程为了获得更好的能量效率而在通用处理器和 C-core 之间跳转执行。因为与通用处理器相比，单独的 C-core 具有明显的能量效率优势，按照 Amdahl's 定律，提高整个系统的能量效率就转变为尽量提高软件在 C-core 上执行的占比，将越来越多的软件调度到C-core上执行。在理想的CoDA架构中，利用更多的硅面积来集成更多的C-core就可以提高负载在专用协处理器上运行的覆盖率，相应地也会提高系统的能量效率。

自动生成 C-core 的工具链按照以下步骤生成 C-cores: 首先，工具链对应用进行 profile 以定位应用中的热代码区域。然后，每一个热代码区域都被分解为基本块或者超块（用来处理 switch 语句）的集合。C-core 采用空间计算方法（spatial computation approach），为块中的每一个操作创建一个特定的功能单元。与此同时，工具链生成 C-core 的控制逻辑。这些控制逻辑使用状态机来控制程序在这些基本块间顺序执行。总之，热代码区域的数据通路和控制逻辑组成了一个单独的 C-core，这个 C-core 的功能通常对应于软件中的外层循环或者一个函数。作为一种优化，C-core 工具链可以将热代码中嵌入的冷代码排除在生成的 C-core 电路之外，这部分代码需要主处理器通过异常处理的方式来处理，例如在定点程序中偶尔出现的浮点操作。因为 C-core 不在算法或功能的角度上修改目标区域代码，所以目标区域代码也可以全部或部分在主处理器上运行。当无法获

得 C-core 时，就会发生程序在主处理上执行的情况。

如图 3-1(b) 所见，C-core 和主处理器使用相同的存储模型，其中主处理器和其他所有的 C-core 共享一级数据缓存。为了减少 C-core 与存储器之间的通信开销，本文 profile 了应用中每个热代码区域对存储器通信的需求，并将这些信息提供给 C-core 选择器和 CoDA 的专用协处理器布局程序。本章所选取的 C-core 设计采用了第二章介绍的使用了选择性去流水化和 cachelet 技术的 C-core 版本[13]，使用这些技术可以在减少面积和能量消耗的同时提高性能。

3.2.2 目标应用

CoDA 中对专用协处理器的选择，依赖于一组目标应用程序。本章研究的目的是为了理解 CoDA 设计是如何扩展的，这种扩展使得原先仅仅包含有限个数专用协处理器的设计扩展到包含成百上千个专用协处理器。相应地本文也需要选择较广泛的目标应用程序。

本文的负载生成程序使用了一系列的“种子”应用，这些初始的应用来源于标准测试程序 SPEC 2006[77], SPEC 2000[78]以及嵌入式测试程序 EEMBC[79]，并修改了它们的部分属性来建模更为广泛的程序特征。表 3-1 列举了这些应用程序以及 C-core 所针对的目标属性。这些初始应用的主要执行部分被 22 个自动生成的 C-core 所覆盖，这些自动生成的 C-core 可以使用标准的 Synopsys 工具生成布局布线后的网标并通过了添加详细寄生参数的门级网表仿真，它们也通过了第五章介绍的 FPGA 验证。

表 3-1 模型中的目标应用

工作集	描述	22nm C-core 面积（mm^2）
astar (SPEC 2006)	path finding	0.044
bzip2 (SPEC 2000)	data compression	0.329
cjpeg (Independent JPEG Group 2000)	jpeg encoding	0.076
crafty (SPEC 2000)	chess	0.580
djpeg (Independent JPEG Group 2002)	jpeg decoding	0.118
gzip (SPEC 2000)	compression/decompression	0.190
mcf (SPEC 2000)	multi-commodity flow	0.056
Viterbi (Embedded Microprocessor Benchmark Consortium 2002)	convolutional decoding	0.039

为了生成更大规模的负载以及使用更多地 C-core 来覆盖这些更大规模的负载，负载生成程序将每一个前面提到的应用复制 2 次、4 次、8 次或者 16 次，并且本文最多调整

生成的 C-core 面积的 50%来模拟热代码密度的变化。通过这种方式，从初始的 8 个应用程序生成了具有 16 个应用，32 个应用，64 个应用和 128 个应用的负载。针对最大的负载情况，系统需要 352 个 C-core 才能够使得专用协处理器获得 97%的代码覆盖率。CoDA 架构为了集成这 352 个 C-core 需要在 22nm 工艺下使用大约 73mm^2 的芯片面积。

3.3 CoDA 模型

本文研究的主要目的是在足够的深度上理解 CoDA 设计中能量消耗是如何分布的，并且理解设计所需要关注的问题。但是由于 CoDA 结构复杂，可以选取的设计参数较多故而存在大量 CoDA 的可行设计和实现，综合并仿真所有这些 CoDA 设计实现是不可能的。所以本章提出了一种分析 CoDA 架构的性能、面积和能量效率的模型，并使用这个分析模型来分析 CoDA 架构。

3.3.1 建模方法学

本文所创建的分析模型由以下三个主要信息驱动：1）使用可以综合的 C-core、CoDA 子部件的属性信息作为模型的输入。这些信息包括单个 C-core 的面积、功耗以及延时信息；Cache 的面积与能量消耗、延时信息；线延时；可以实现的门控电源的效率等等；2）使用 trace 的技术对负载中程序的动态执行行为进行分析，并输入本文的分析模型。这些信息包括线程如何迁移，Cache 一致性的影响，Cache 命中/缺失等各种信息；3）搜集了前人关于 C-core 和这些负载的信息，并将所有的 45nm 或者 40nm 工艺下的参数换算到 22nm 工艺。本文的模型计算结果可以提供某一种 CoDA 配置下，能量消耗、面积以及性能参数等方面的分布情况。

本文使用 Synopsys 的 Design Compiler、IC Compiler 和 PrimeTime 来评估针对表 3-1 中的应用进行定制的 C-core。这些 C-core 都是可以综合、布局布线的，并且团队已经基本完成了流片工作。这些构件好的 C-core 信息都是独立输入分析模型的。另外本文作者也获得了通用主处理器布局布线后的网表，并从中提取功耗、面积以及延时信息，并将这些信息输入到分析模型。以上这些信息的提取都是使用台积电（TSMC）45nm 通用工艺节点和 40nm 低功耗工艺节点获得的，本文的模型将这些信息换算到 22nm 所对应的各种参数。

为了分析程序动态执行的行为，本文使用了 LLVM[80]来标记每一个执行模块，然后建立不同执行部分在专用协处理器上执行还是在通用处理器上执行的映射。在 CoDA 架构中，每一个执行模块模拟了一段代码被映射到某一个特定的瓦片，并在这个瓦片的通用处理器或者专用协处理器上执行。每一个被标记的执行模块提供了详细的动态执行信息，这些信息包括：存储器操作信息（一致性消息，Cache 到 Cache 的数据迁移），程序段在片上网络的迁移信息（程序在 C-core 和非 C-core 之间的迁移，程序在瓦片间的迁移）。除了以上这些程序运行的开销，本文的模型中进程可以每时钟周期执行一条指令。本文仿真这些被标记过的二进制代码获得以上详细的动态执行信息，并将这些信息输入

到分析模型中。

3.3.2 模型参数

$$Wire\_length = 2\sum_{i=1}^{n}\sqrt{Component\_Area_i}, \quad (3\text{-}1)$$

$$i \in every\ component\ along\ the\ path.$$

本文的分析模型中有些参数是可以从其他文献中直接获得，另外一些参数是在40nm 或 45nm 工艺下进行评估然后换算到 22nm 工艺。表 3-2 列出了关键的参数。本文将每毫米互连线所消耗能量与线的长度相乘，来计算互连线所消耗的能量。公式（3-1），用于计算互连线的长度，因为 ASIC 实现中仅有水平和垂直的金属互连线资源，所以本文计算线长度时采用的是两个模块间的曼哈顿距离（Manhattan-distance）。为了计算从专用协处理器到 L1 数据高速缓存多选器的线长度，本文将专用协处理器按照对存储的需求进行排序，并将对存储需求高的专用协处理器布局到距离多选器较近的位置。

在分析模型中，本文进行了以下假设。1）通过在综合后的 C-core 上仿真执行测试程序所进行的评估，本文假设在同一个瓦片内程序执行从软件执行的主处理器跳转到硬件执行的专用协处理器需要 30 个时钟周期。2）基于 MIT Raw 处理器的评估[72]，本文假设程序在激活的瓦片之间迁移需要 300 个时钟周期，该时间包括了异常处理和上线文通过片上网络进行迁移所需要的时间。

表 3-2 模型中的参数值

模型参数	参数值来源
每毫米的线能量消耗	Bill Dally 2009年DAC报告
主处理器能量消耗	主处理器平均每条指令的能量消耗来自于文献[6]，然后使用公式（3-3）和（3-4）换算到22nm。
专用协处理器能量消耗	C-core等效的平均每天指令能量消耗来自于文献[6]，然后使用公式（3-3）和（3-4）换算到22nm。
NoC路由能量消耗	模型中假定每一次路由决策所需能量消耗等于专用协处理器执行一条等效指令的能量消耗。
Cache的漏电功耗，面积和访问时间	CACTI[81]，并使用公式换算到22nm。
主存带宽	本文假定系统中使用LPDDR2，带宽为3.2GB/s。
晶体管速度	高性能库（HP）和低静态功耗库（LSTP）相对的延时来自于对综合后电路的评估，类似的值也可以从ITRS的数据计算得出。模型假定了低功耗库(LP)的延迟是其他两个库的均值。所以模型中：HP库相对延时假设为1，LP库为1.25，LSTP库为2.5。
晶体管漏电功耗和动态功耗所需能量	使用可综合电路评估，在换算到22nm。
软件与硬件切换时间	30周期，使用测试程序评估
在激活的瓦片间执行迁移时间	300周期，使用测试程序评估

$$Area_{new} = Area_{old} * (\lambda_{new} / \lambda_{old})^2 \quad (3\text{-}2)$$

$$LEnergy\_per\_square\_mm_{new} = (LEnergy\_per\_suqare\_mm_{old} * 3D\_Facter * (\lambda_{old} / \lambda_{new})^2) \quad (3\text{-}3)$$

$$Dynamic\_energy_{new} = Dynamic\_energy_{old} * (\lambda_{new} / \lambda_{old}) \quad (3\text{-}4)$$

公式（3-2）、（3-3）和（3-4）用于将 45nm 和/或 40nm 下所获得的面积、漏电功耗以及动态功耗换算到新工艺下。其中 λ_{old} 和 λ_{new} 分别表示老工艺的特征尺寸和新工艺的特征尺寸。公式（3-2）比较直观，对于计算资源限制的设计来说（相对于 IO 限制）芯片上晶体管的密度随工艺的进步不断提高。公式（3-3）稍微复杂一点。晶体管漏电功耗参数其实是设计者可控的，设计者可以通过调节阈值电压和供电电压来改变这个参数。然而，在登纳德缩放模型失效后，设计者都最求尽量限制漏电功耗。因此公式（3-3）首先假定每个晶体管具有固定的漏电功耗。为了考虑 32nm 到 22nm 晶体管演进所引进的平面晶体管向 3D 晶体管(FinFET)转变，本文引入了一个额外缩放参数：3D 参数。本文的模型中 3D 参数的值取为 0.7，该值是从 Intel 公司关于 22nm 工艺的文献[82]中计算得出的。第三部分是晶体管密度变化参数。本文的参数选择平均每平方毫米的漏电能量，是因为模型也需要这个参数来计算非激活状态电路的漏电能量，使用该参数计算较为方便。对本模型中所考虑的三种不同类型晶体管，还要单独计算每种类型晶体管平均单位面积的漏电能量。最后，公式（3-4）是比较直接的动态能量模型。在暗硅时代，无法通过降低电压的方式来节省芯片能耗，单个晶体管动态能耗的节省仅仅直接来自于电容的降低。电容的降低表现在公式（3-4）的第二个因子上。

本文使用仿真器进行系统仿真，并用 Trace 的方法获得程序对一级 Cache 和二级 Cache 的仿存数据。为了对高速缓存的面积、动态功耗和静态功耗等物理特征进行建模，本文使用了 CACTI[81]。本文在 CACTI 中选择 32nm 工艺对高速缓存建模，然后使用公式将参数换算到 22nm。对面积、漏电功耗和动态能耗的换算分别使用前述公式（3-2）、（3-3）和（3-4）。

由于静态功耗在系统中所占比例较高，为了探讨极限情况，需要尽量减少高速缓存在系统中的面积。本文假设每一个二级高速缓存都是非包含的，并且二级缓存仅仅供同一个电压域内的一级高速缓存使用。被一级缓存逐出的数据将被分配到二级缓存。在二级缓存命中的数据将被迁移到相应地一级缓存，并将二级缓存的该数据行置为无效。当一级缓存缺失时，需要首先询问同一电压域内的二级 Cache 的一致性目录，如果确定内容在其他一级缓存，目录将发起一次高速缓存到高速缓存的数据传输，以便在两个一级缓存间传输数据。

被标记区域的动态和静态指令数量可以被用来对没有进行布局布线的 C-core 的面积和覆盖率进行建模。静态指令数量是程序段的指令数量，而动态指令数量是程序执行时动态执行的指令的数量。对于每一个单独的 C-core 来说，他们的面积从 0.0015mm^2

到0.28mm^{2}不等。对这些没有进行布局布线的C-core面积的评估，基于对生成这些C-core的软件进行的静态指令分析，这种分析与分析 SiChrome 类似。针对每一种不同的操作，进行面积评估然后再相加，例如加法、乘法、移位等指令会相应地生成加法器、乘法器、移位器等。本文将这部分模型与之前发表的论文[13]中进行了布局布线的 C-core 数据进行了校准。同样，本文也使用了论文[13]中对一级仿存延时对程序执行时间影响的数据校验了分析模型。一级仿存延时的增加是由于在较大的瓦片中，集成大量的 C-core 导致有些 C-core 被放置在离一级缓存较远的位置。

工艺库的选择对晶体管的漏电功耗和性能有较大的影响。本文评估布局布线后的 C-core 使用的是台积电 40nm 低功耗工艺库和 45nm 通用工艺库。使用这些数据，本文对 C-core 以及非存储逻辑进行评估，然后换算到 22nm，所得到的静态功耗优化、低功耗和性能优化工艺库的功耗分别是 0.25mW/ mm^2, 1.02 mW/mm^2, 和 29.28 mW/mm^2。换算使用公式（3-4）。为了对不同工艺库对性能的影响进行建模，本文计算了 CoDA 采用不同工艺库的延迟情况，具体的参数从 ITRS 的报告中获得。模型中对于静态功耗优化、低功耗和性能优化的工艺库所采取的性能系数分别是 2.0、1.0 和 0.8。

模型中假定本文使用的高能量效率的顺序执行主处理器运行在 3GHz。根据论文[6]中关于 C-core 能量消耗的分布，本文建模 C-core 中非存储计算操作的每条指令消耗的动态功耗比使用的主处理器低 30 倍。评估得到主处理器在 22nm 工艺下每条指令需要 43pJ 的能量。文献[13]中表明 C-core 的执行速度平均比主处理器快 27%。

3.3.3　性能、面积和能耗模型

$$
Performance \begin{cases} Software\ execution\ cycles\ on\ host\ processor \\ Hardware\ execution\ cycles\ on\ coprocessors \\ Software\ and\ hardware\ transition\ cycles \\ Tile\ transition\ cycles\ including\ interrupt\ handling\ and\ context\ transfers \\ NoC\ route\ cycles \\ L1\ cache\ access\ cycles \\ L2\ cache\ access\ cycles \\ Main\ memory\ access\ cycles \\ Cache-to-cache\ transfer\ cycles\ including\ all\ coherence \\ Cycles\ per\ instruction\ (CPI) \end{cases}
$$

图 3-3 性能模型的信息组成

$$
Area \begin{cases} Area\ of\ the\ tiles \begin{cases} Host\ processor\ area \\ L1\ cache\ area \\ NoC\ router\ area \\ Coprocessors\ area \\ MUX\ area \end{cases} \\ L2\ cache\ area \end{cases}
$$

图 3-4 面积模型的信息组成

分析评估模型的输出主要有三个：性能、面积和能量。本小节分别对这三个方面进行更为详细的介绍。图 3-3 展示了分析模型中所建模的有关性能的各个因素。除了 CPI 以外，其他的所有关于性能的值都直接来自于仿真器。之所以 CPI 是单独计算的，是因为 CPI 与以下因素有关：瓦片内处理器的布局，例如其他应用所需的 C-core 可能会获得更高的存储优先级所以放置在与存储更近的地方，以至于这个专用协处理器到 L1 的距离更远，延时变长，而这个信息来自于系统建模层次。在本文的分析模型中，使用所有这些处理器部件的执行时间来计算有效的 CPI，有效的 CPI 被用来计算每条指令所消耗的能量。

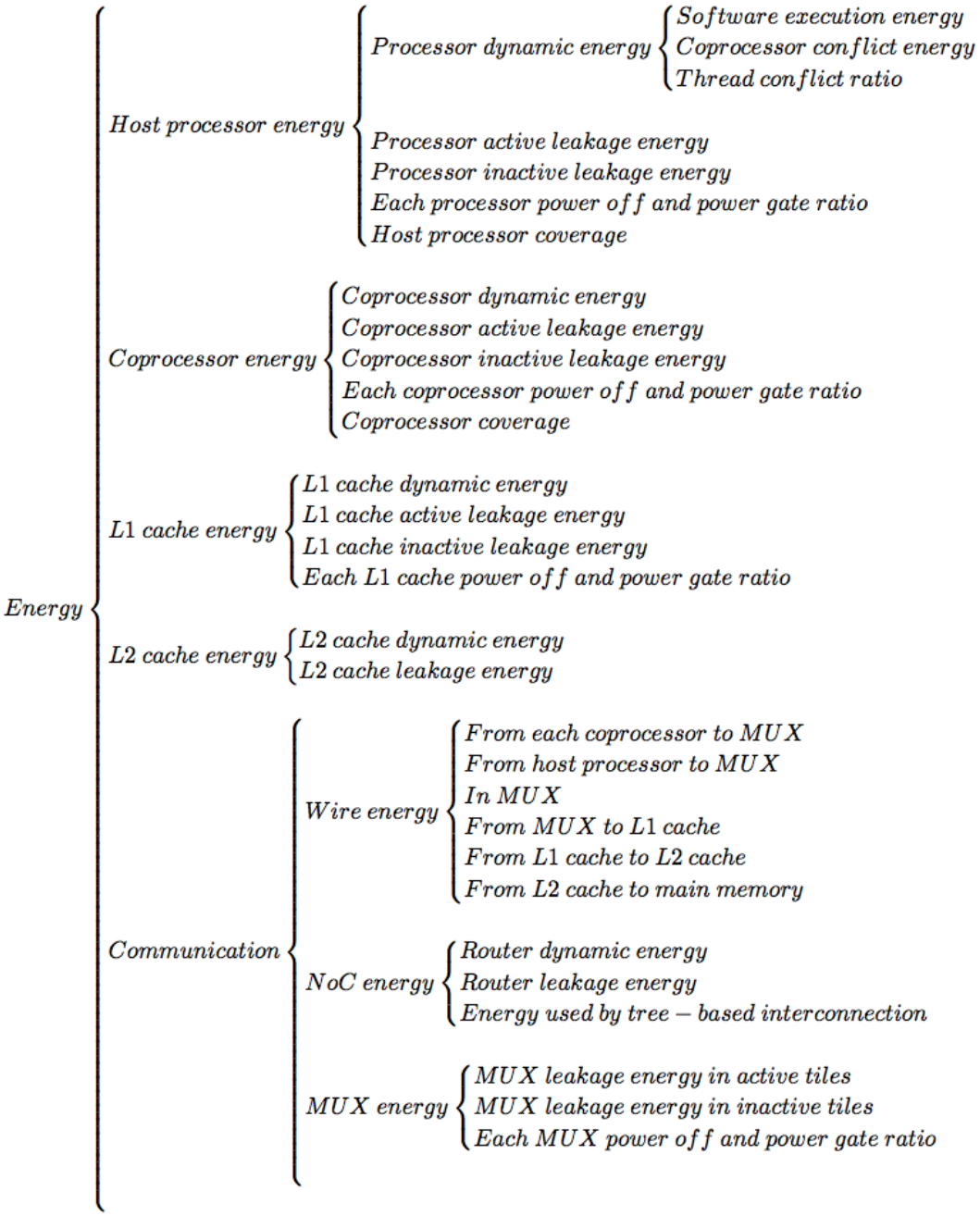


图 3-5 平均每条指令所消耗能量模型的信息组成

图 3-4 列举了芯片面积的主要组成部分。在本文的分析模型中，L2 高速缓冲的总面积不但取决于 L2 高速缓存的大小，也取决于电压域的个数。因为，本文的系统中每一个电压域都包含一个独立的 L2 高速缓存。

分析模型在多个层次上对不同器件的能量消耗进行了建模。为了对不同测试程序进行归一化，以及考虑到硬件执行和软件执行，本文将能量模型中能量的消耗归一化为每一条等价软件指令所消耗的能量。图 3-5 展示了能量模型中每一个建模部件以及他们的层次关系。下一章，在模型中增加了并发执行时竞争使用专用协处理器对能量的影响。

本文主要关注基于 CoDA 架构中片上部分的能量消耗情况。在实际的系统中，能量的消耗还包括其他部分，其他一些文献已经或正在研究优化这些部分。例如，在研究低功耗 DRAM 的前景时，研究人员采用了硅通孔（through-silicon vias），package-on-package 和低功耗片外信号技术。尽管这些技术可以很容易和本文的研究结合起来，但是这些技术最终的特性仍不明朗。为了保持研究的独特性和准确性，本文暂时不讨论这些方面，这些内容暂且由其他人进行研究[83, 84]。

3.4 CoDA 架构参数空间

图 3-1 所展示的基本体系结构在 CoDA 架构的配置方面还有很大的灵活性。有效的 CoDA 设计还可以包含不同数量的瓦片，可以改变的瓦片大小，不同的 Cache 配置等等。此外，不同设计约束条件下（例如，不同面积预算，功耗预算和性能要求等）所产生的配置也不同。

为了后面理解不同 CoDA 架构的优化设计，在这小节中对 CoDA 设计参数进行系统的讨论。首先详细列举了系统中所有考虑的不同设计参数以及参数值，之后分析了各个参数对 CoDA 模型（面积、性能和能耗）的具体影响。

3.4.1 设计参数

表 3-3 CoDA 设计空间参数

参数	参数值
工作负载大小（应用数量）	8，16，32，64，128
L2 Cache大小 （KB）	512，2048，8192
每个瓦片中L1 Cache大小 （KB）	8，16，32，64
最大瓦片面积（mm^2）	0.5，2，8，32，无限大小
电压域个数	1，4
门控电源效率	0，90%，95%，98%
晶体管库	LSTP，LP，HP

通过组合不同的设计参数，可以潜在的生成成百上千的不同 CoDA 设计。为了理解

不同设计参数选择的影响，本文系统的研究了面向多种不同负载规模的可能配置的参数选择，这也就是 CoDA 架构的参数空间。

表 3-3 列举了本文所考虑的 CoDA 设计空间参数。选择不同的设计空间参数可以产生不同的 CoDA 设计，包括传统的具有较大容量 Cache 的通用处理器并集成少量的专用协处理器，也包括具有大规模瓦片结构众核处理器并集成大量专用协处理器的设计。为了限制设计空间的大小，本文不考虑瓦片之间包含不同 Cache 配置资源的情况。因此总的来说，本文针对不同负载评估了总共 7200 种不同的 CoDA 配置。

3.4.2 参数与模型

表 3-4 不同参数对系统能量消耗，面积和性能模型的影响

参数	直接影响	对最终结果的影响
工作负载大小	专用协处理器数量	芯片的面积，漏电功耗，通信距离
Cache大小	芯片面积	激活区域的漏电功耗，非激活区域的漏电功耗和动态功耗
	互连线长度	互连线消耗的能量
	Cache缺失率和主存访存率	CPI和执行时间
	命中延时	CPI和执行时间
最大瓦片面积	芯片面积	激活区域的漏电功耗，非激活区域的漏电功耗和动态功耗
	互连线长度	互连线消耗的能量
	瓦片个数	L1 Cache数量，线程冲突率
电压域个数	L2 Cache的数量	芯片面积和漏电功耗所需能量
	片上和片外互连线长度	互连线消耗的能量和NoC所消耗的能量
门控电源效率	非激活区域的漏电功耗	CoDA，Cache，主处理器，MUX等器件消耗的能量
晶体管库	每条指令需要的静态和动态功耗	所有器件消耗的能量
	CPI	非存储器操作执行的总时间

表 3-4 描述了每一个具体地参数是如何影响第三节中所描述的模型的。有几个参数对模型的影响是非常直观的。对于每一个本文所探讨的负载大小，维持专用协处理器可以覆盖 98%的程序执行，这样随着负载规模的增大，需要调整覆盖所需的专用协处理器的数量。设计空间包括几种不同的 L1 和 L2 高速缓存配置。不同高速缓存的大小会直接影响芯片的总面积、专用处理器到共享高速缓存的互连线长度以及高速缓存的性能等。高速缓存所带来的间接影响也比较直观。瓦片面积参数描述了瓦片化设计中单个瓦片的最大面积。较大的瓦片可以包含更多的专用协处理器并减少瓦片之间的跳数（因为瓦片数量少），但是瓦片内部的 C-core 可能具有较长的通信路径（因为专用处理器数量多）。

芯片面积会影响 CoDA 芯片中激活区域和非激活区域的漏电功耗；互连线长度直接影响了互连线的能耗；由于 CoDA 芯片中每一个瓦片中都有一个 L1 高速缓存，所以瓦片个数决定了 L1 高速缓存的数量；瓦片个数也决定了 CoDA 架构中同时执行的线程数量，也就间接决定了线程之间冲突的频率。

最后三个参数较为复杂。其中有两个是关于电源管理的参数：芯片上独立电压域的数量和关断电路电源时的效率。芯片外的电压调节器就可以将与之相连的电压域电源关断，这样可以将漏电流减少到最低。“门控电源效率”决定了供电的电压域中通过门控电源方式将电路变为非激活电路时候的效率。设计者可以在一定程度上控制这些参数（例如，使用最先进的门控电源电路[73]），但是这个参数也取决于加工工艺。本文建模的 CoDA 架构中每一个电压域都包含一个共享的二级 Cache，所以电压域数量直接影响了二级 Cache 的数量。最后一个参数决定了实现设计所使用的标准单元库。可用的选项包括：低静态功耗库（LSTP），低功耗库（LP），和高性能库（HP）。第 3.3 小节描述了与这些库相关的功耗和性能参数。

3.5 小结

大规模 CoDA 架构支持大规模、多样的应用程序负载集合。为了使得 CoDA 设计对这样的应用具有较强的适应性，本章首先描述了多维可扩展的 CoDA 架构及其能耗管理策略。本文所探索的 CoDA 可以由不同数量的瓦片组成，瓦片内部可以集成数量不同的专用协处理器并且每一个专用协处理器都可以是异构的。CoDA 设计的能耗管理方法包括电压域、门控电源以及门控时钟。

由于 CoDA 架构较为复杂并且设计空间巨大，完全使用硬件实现之后进行评估的方法不能有效对设计空间进行探索。因此本章对 CoDA 架构进行了建模，并较为详细地介绍了模型的目标硬件结构，建模方法学，以及各种模型参数。本文的模型包括三个主要部分：性能，面积和能量消耗。这些都是下一章探索 CoDA 设计空间的基础。

本章的主要内容发表于论文“Exploring Energy Scalability in Coprocessor-Dominated Architectures for Dark Silicon”。

4 CoDA 能效研究

随着 CoDA 架构集成越来越多的专用协处理器，集成开销也会不断增长，这给 CoDA 设计有效扩展带来了严重挑战。这些开销是否会吞噬大量使用专用协处理器带来的能量效率优势，以及大规模 CoDA 设计与同构多核处理器相比是否还有能量效率优势是本章探索的重点。只有当大规模的 CoDA 设计仍然比同构多核处理器具有明显的能量效率优势，CoDA 设计的扩展才是有效的。

本章将系统地探索 CoDA 架构的设计空间，仔细地观察 CoDA 架构如何有效地在大规模以及存在大量并发执行的负载情况下进行扩展。本章仔细地探索了 7200 种不同 CoDA 设计，以便理解 Cache 大小、瓦片大小等等高层次的结构参数以及功耗管理策略、晶体管类型等等底层实现参数对 CoDA 设计能量效率的影响。

本章还研究了当驱动 CoDA 架构生成的目标工作负载和实际工作负载不匹配时导致专用协处理器数量、分布错配，并造成同时运行的多个程序竞争使用特定专用协处理器时对 CoDA 能量效率的影响，以及缓解这种损害的有效方法。

本章的实验结果表明，支持大规模负载的 CoDA 设计，多个线程可以共同分担集成大量专用协处理器所带来的开销，这样可以使执行每条指令所消耗的能量减少 3.7 倍；而对于小规模的设计可以减少 5.3 倍的能耗。因此大规模 CoDA 架构与传统同构多核处理器相比还是可以成倍提高能量效率的，这也证明了 CoDA 架构可以有效扩展。

本章还讨论了大规模 CoDA 设计是否会对芯片加工的良率产生影响以及并发执行时 CoDA 是否对片外访存造成更大压力。

本章的主要内容如下：第一小节先列出了本章研究 CoDA 能耗问题的贡献；第二小节探索了 CoDA 在不同设计空间参数下能量消耗的分布；第三小节解决了并发执行相关的问题；第四小节列举进一步降低 CoDA 能量消耗的潜在方法；第五小节为相关研究情况；第六小节对本章进行了总结。

4.1 CoDA 能效研究发现

本章表明在大规模 CoDA 架构中，效率的主要限制来源于芯片上除专用协处理器外其他部件的能量效率以及暗硅部分的漏电功耗。本章的研究具有以下几点发现：

1）如果没有激进的功耗管理机制（电压域，门控电源等），漏电功耗将吞噬大规模 CoDA 所带来的优势。本章还指出，即使有激进的功耗管理机制，漏电功耗所占总功耗的比例仍然伴随专用协处理器数量的增加而增加。在 CoDA 架构中，闲置部件的漏电功耗可以超过激活部件的动态功耗。尽管如此，本章的实验结果表明当负载需要集成几百个专用协处理器的时候，CoDA 架构仍然可以带来 3.7 倍的能量效率优化。

2）CoDA 架构中必须采用高能量效率的功耗管理、互连网络和存储系统，这样才

能保证在 CoDA 架构扩展时仍然能获得较高的全局能量效率优化。本章的实验结果指明了改善功耗管理、互连网络和存储系统等因素，与改善 CoDA 能量效率之间的关系。特别地，实验结果为未来研究如何使暗硅处于完全没有能量消耗的状态提供了强有力的动机 。对于并发执行的负载，实验结果表明线程竞争特定专用协处理器所造成的影响可以有效的缓解，代价是仅仅增加一点芯片面积。

3）可有效扩展的 CoDA 架构对较大的负载仍然可以显著地降低能耗。实验结果表明，对于较轻的负载，CoDA 设计可以带来 5.3 倍的能量效率优化，并带来 5 倍的能量延时积（energy-delay product, EDP）优化。对于运行上百个应用的较重负载，CoDA 可以带来 3.7 倍的能量效率优化，并带来 3.5 倍的 EDP 优化。

4.2 CoDA 设计空间探索

CoDA 架构设计复杂，并且配置方面具有较大的灵活性和多样性。为了理解不同 CoDA 架构的优化设计，本小节对 CoDA 设计空间进行系统的探索。本节使用第三章所描述的应用负载来驱动生成大量不同的 CoDA 设计。这些设计包含了仅仅具有一个瓦片和几个 C-core 的简单设计，还包括面向大规模负载的具有几十个瓦片和几百个 C-core 的众核可扩展复杂 CoDA 设计。

在获得特定设计的基础上，使用第三章所述的分析模型对设计的面积、性能和能耗等进行仔细地分析，再对所有设计的结果进行统计分析，分析的结果见下文。此外还讨论了应用规模与集成代价之间的关系，以及 CoDA 设计对缺陷率的影响。

4.2.1 CoDA 能效帕累托分析

图 4-1 展示了设计空间研究的结果，这些结果产生于特定的门控电源效率以及特别数量的电压域下，本文考察的设计分别包含一个电压域和 4 个电压域的情况，门控电源效率关注了 0%，90%，95%和 98%。但是由于 90%，95%和 98%的趋势类似，为了使图形便于观察，所以仅仅保留了 0%和 98.1%两种情况。图 4-1（a）画出了本文探索的支持所有不同负载的单电压域设计。横坐标为设计所使用的芯片面积，纵坐标为相对的能量延时积（EDP），比较的基准为不包含专用协处理器并使用 32KB L1 高速缓存和 512KB L2 高速缓存的通用处理器。图 4-1（b）与 4-1（a）相似，但是采用了最多 4 个电压域。图（a）和（b）中每一个点是一种负载在某一个特定参数组合下设计的评估结果。图片用不同颜色标记了不同的门控电源效率，也用不同颜色标记了工作集的大小。图 4-1（c）到 4-1（f）分别针对不同的应用规模（8，32，64 和 128）[1]画出了能量和延时关系的四组极限情况下的帕累托边界（Pareto-frontier）曲线，这些曲线是针对每一种规模负载采用不同的门控电源效率和电压域组合生成的，不同的组合标示为 X% PGE, Y VD 的形式（X 为门控电源效率，Y 为电压域个数）。这些帕累托边界描述了针对某一设

[1] 应用规模为 16 的情况与规模为 32 的情况曲线极为类似，为了不分散读者注意力所以这里没有给出 16 个应用的曲线。

计没有其他设计可以不但更快（延时更小）并且更有效率（平均每条指令消耗更少能量）。换句话说，帕累托边界上的点为该条件下的最优解。

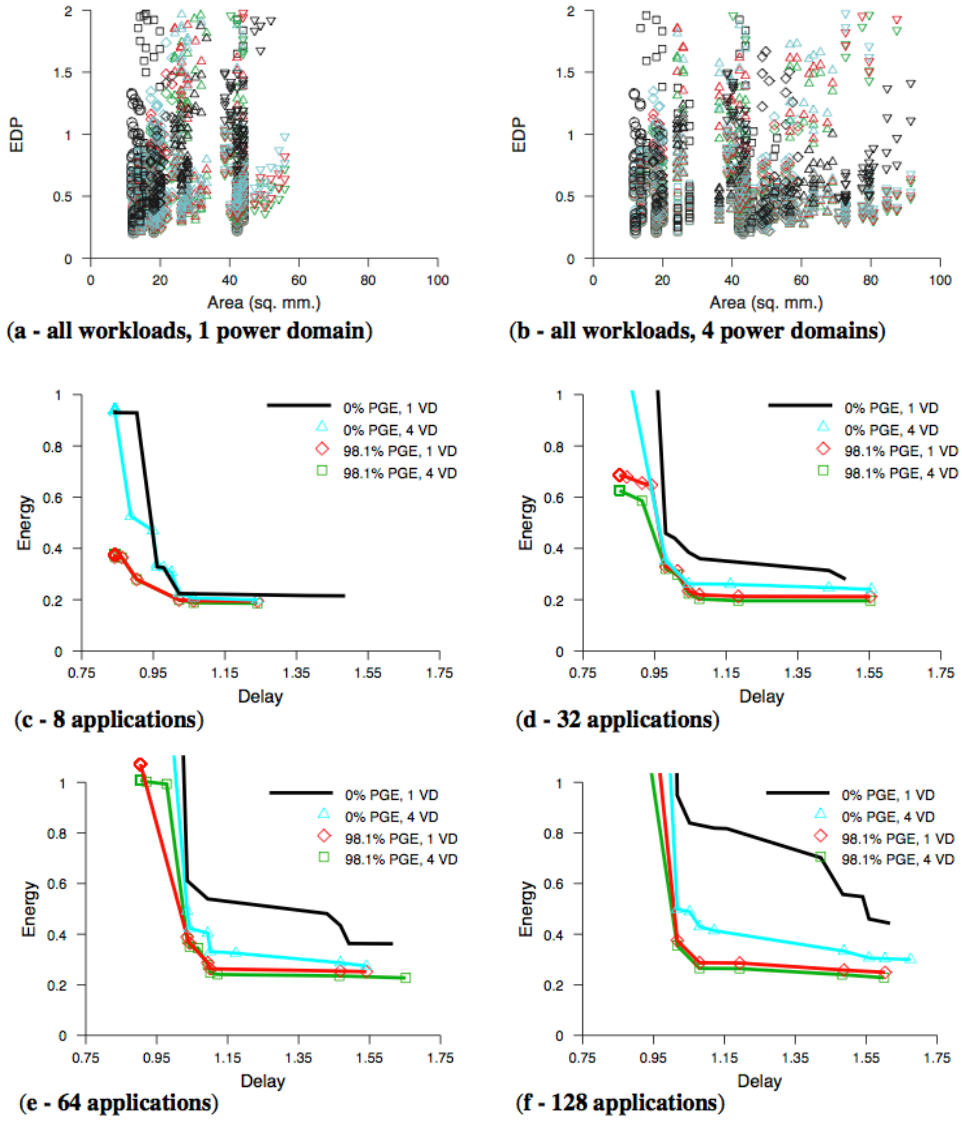


图 4-1 功耗延时积(EDP) vs. 面积以及能量 vs.延时关系

从图 4-1（c）-（f）中可以清晰的看出没有功耗管理的黑线和具有多电压域却没有门控电源的蓝线与具有功耗管理的红线和绿线之间的距离随应用规模增长而越来越大，这说明功耗管理的影响随着 CoDA 架构规模的增大而增加。如果没有激进的动态漏电功耗管理，没有门控电源的大规模的设计必须采用低静态功耗的晶体管来减少能量开销，这样牺牲了性能。这在图形上表现为随应用规模的增长，获得同样的能量开销，没有功耗管理的设计的延时越来越大。较大的延时说明，此时的设计采用了低静态功耗的晶体管。与之类似，使用高性能高静态功耗晶体管来获得高性能，即使使用门控电源，当应用负载规模增大时能量消耗也快速增加。这在图形上表现为随应用规模的增长，图形的左上角红线和绿线变得越来越陡峭。在 CoDA 架构上这是因为为了保证专用电路的覆盖

率，应用增加时就要增加更多地专用协处理器，这些专用协处理器带来了相应的漏电功耗开销。

即使具有多个电压域，对于大规模负载来说，瓦片的粒度或者说门控电源的粒度对于系统也是至关重要的。虽然增加电压域会增加 L2 高速缓存的个数，但是却可以缓解对门控电源实现的依赖。这些增加的 L2 高速缓存面积的开销体现在图 4-1（a）与 4-1（b）横坐标（面积）的分布上。增加电压域也可以使片上的门控电源可以在更粗的粒度上进行操作，这可以进一步减少门控电源的复杂性。从图 4-1（e）和 4-1（f）中的蓝线和绿线之间的距离以及黑线和红线之间的距离，可以明显看出门控电源的影响。

图 4-1（c）到 4-1（f）表明，改变电压域数量对帕累托曲线（Pareto curve）的右下角影响有限（右下角各种曲线之间距离与中间段相比较近），这是因为右下角的设计实现已经采用了低功耗的晶体管和有效的门控电源。电压域数量对左上角高性能的设计影响较大。整体上，采用多电压域的设计具有更高的效率，并且设计也需要更多地芯片面积来获得这种优势。在芯片上具有不断增加暗硅的时代，这将是一个合理的折中。选取不同门控电源技术所带来的优势几乎被增加的电压域所掩盖，设计中即使没有细粒度的门控电源仍然可以提高三倍的能量效率。但是在负载应用较重的情况下，是否采用先进的门控电源将对能量效率带来稍大影响。对于单电压域设计情况，当应用负载规模较大时，CoDA 设计必须采用激进的动态功耗管理技术，否则大量专用协处理器所导致的漏电功耗将降低系统的整体能量效率。虽然 L2 高速缓存复制不能无限扩展，但是这些趋势表明对于比本文所观察的负载更大的负载来说，应该将设计分割为更多的电压域来提高效率直到复制 L2 高速缓存的开销超过了芯片的面积约束。

表 4-1 列出了针对每种不同应用规模时，能量延时积最优化时 CoDA 架构所采用的设计参数。能量延时积最优解对大部分应用规模都是类似的，但是对于最大规模的设计需要采用较大的瓦片来减少程序在瓦片间迁移时的传输时间。这也是本文多维可扩展架构所带来的优势，可以更加灵活地适应各种应用环境。

表 4-1 能量延时积最优化设计

负载中的应用个数	L2 Cache (KB)	L1 Cache (KB)	最大瓦片面积 (mm^2)	瓦片数量	C-core 数量	EDP vs SW	面积 (mm^2)	pJ/inst	加速 vs SW	工艺库	瓦片/电压域
基准	512	32	-	1	0	1.0	10.29	51.05	1.0	HP	1
8	512	32	0.5	5	22	0.200	43.20	10.40	0.941	LP	1-2
16	512	16	0.5	9	44	0.206	45.20	9.97	0.857	LP	2-3
32	512	32	0.5	20	88	0.218	50.70	11.50	0.930	LP	5
64	512	32	0.5	40	176	0.270	60.70	13.39	0.892	LP	10
128	512	32	2.0	16	352	0.286	72.70	14.80	0.926	LP	4

4.2.2 CoDA 规模与集成代价变化趋势

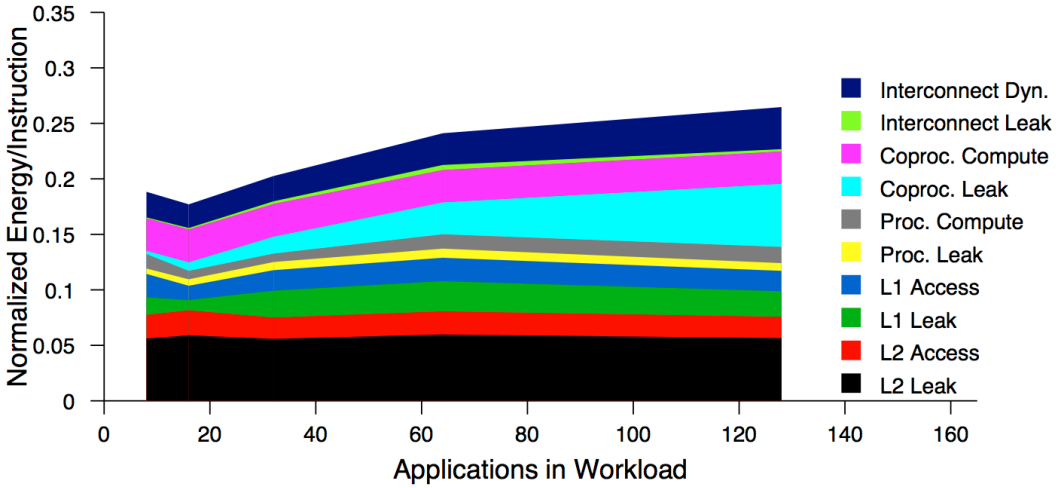


图 4-2 EDP 最有设计的能量消耗分布

图 4-2 展示了上一小节中当应用负载数量逐渐增大时，设计空间中特定规模负载下最优化的 CoDA 设计中总能量消耗的各个组成部分。趋势表明，对于大型设计（覆盖更大的应用），漏电功耗和互连占据了重要部分，而计算所需能量占比逐渐降低，这是因为大部分计算都迁移到高能量效率的专用协处理器上。结果表明，尽管 CoDA 设计采用了足够多的 C-core 来覆盖应用负载并且可以降低程序运行时 94%的用于计算的能量消耗，但是计算所需能量在总能量消耗中仅占很小一部分。大部分的能量都被暗硅消耗。即使采用了多个电压域和效率为 98%的门控电源[73]，总能量中将近有 50%的能量还是被漏电功耗所占据。数据在局部和全局互连上的传输以及对 L2 高速缓存的访问也占据了很大一部分能量消耗。互连线所使用的能量也超过了运算逻辑所需要的能量。

从图 4-2 中，可以认识到以下两点。第一点：尽管大家知道像 CoDA 这样的大规模复杂芯片，设计时需要管理漏电功耗；但需要注意的是，对已经采用了多个电压域以及效率为 98%的门控电源后，非激活器件的漏电功耗仍然不容忽视。第二点：较为幸运的是，尽管从图 4-2 中可以看出随着 CoDA 架构的扩大，各种开销不断增长。但是对于大规模的 CoDA 仍然可以节省几倍的能量消耗，也就是说 CoDA 架构确实是可以有效扩展的，这是较为乐观的结论。数据表明尽管存在一些开销，但是可以覆盖 128 个应用的 CoDA 架构与通用处理器相比仍然可以获得数倍的能量效率提高。对于很多应用环境，例如 Android 平台，128 个应用几乎可以覆盖 80%的系统运行时间[3]，这意味着 CoDA 的扩展能力可以充分满足实际的需求。

4.2.3 CoDA 规模与缺陷率

CoDA 规模扩大相应地芯片面积也会随之增长。芯片面积扩大所带来的潜在缺陷率上升是一个有趣的值得讨论的话题。因为原始的应用程序可以运行在未经修改的任何一个瓦片中的通用主处理器上，并且在通用主处理器上运行并不需要特定的 C-core 支持，

所以 CoDA 的设计研究对于大部分片上模块来说是可以容忍缺陷的；对于有缺陷的部件，简单地不再去使用即可。事实上，用来检测 C-core 是否可获得以避免冲突的机制可以直接用在避免使用有缺陷的部件上。因此，CoDA 可以在一定程度上缓解日益升高的缺陷率问题，或者当某一目标应用从负载中移除时，CoDA 芯片在功能上还是可用的，并且还可以降低能量消耗。因为 CoDA 设计中所使用的 C-core 对程序员来说是透明的，所以程序员在编写程序时并不会有可用性或者虚拟化方面的障碍。因此，大规模的 CoDA 即使无缺陷的良率低于较小的 SoC 芯片，但是有缺陷的 CoDA 芯片上的大部分器件还是可用的。

4.3 CoDA 和并发执行

在当今从智能手机到数据中心的计算平台上，并发执行无处不在，所以理解多线程和多应用程序负载对 CoDA 的影响尤其重要（反之亦然）。本小结讨论多线程对 CoDA 架构积极的影响和消极的影响，并介绍几种技术来解决多线程所引入的新问题。

从积极的角度来看，CoDA 架构中同时运行多个线程可以提高全局的能量效率。这是因为他们可以分摊基本固定的以下能量开销：1）漏电功耗引起的能量开销；2）互连网络的开销；3）存储系统开销。但是同时运行的并发程序可能会竞争使用某些特定的 C-core。当两个应用程序同时访问专门为共享的库函数（例如：glibc）而设计的 C-core 以及当多个线程执行相同的应用并执行到同一个函数的时候，他们就会竞争 C-core。在这种情况下，竞争失败的线程或者返回通用主处理器上执行，这样会牺牲能量效率；或者等待，直到可以使用 C-core，这样会牺牲性能。本研究中假设调度器总是调度竞争失败的线程返回通用主处理器上执行。更加激进的调度器可以采用更为复杂的启发式技术来动态地决定是牺牲能量效率还是牺牲性能。

4.3.1 CoDA 对目标负载的敏感性

CoDA 架构中，竞争 C-core 这种情况发生的次数取决于某种 C-core 在系统中实例化的个数以及需要使用这个 C-core 的线程的数量。用于 Profile 应用热代码的程序也可以用于获取 C-core 使用情况的信息。这些信息使作者知道在特定应用负载集合下系统中需要复制多少个特定的 C-core 来避免冲突。

如果 profile 出来的数据与实际负载动态运行的需求不同，多个同时运行的线程所带来的共同分担固定开销的好处也会受到损害。在本文的并发执行实验中，考虑了两种不同的负载分布情况。第一种情况是负载均衡分布，该种情况描述了所有应用程序在系统中都占用相同的执行时间；第二种负载分布情况是非均衡分布，这种情况描述了 10%的应用占用了系统中 90%的运行时间。

图 4-3 和图 4-4 展示了程序竞争专用协处理器和负载与专用协处理器类型不匹配时的能量效率。实验首先选择某一特定 CoDA 设计；被选取的是工作集为 128 个程序时，并且程序负载均匀分布条件下设计空间中可以提供最佳能量延时积的设计。当负载均匀

分布时，设计所导致的程序竞争专用协处理器概率较小。表 4-1 的最后一行是这个设计的具体参数。这个 CoDA 设计包含 16 个瓦片，最多可以支持 16 个同时运行的线程。

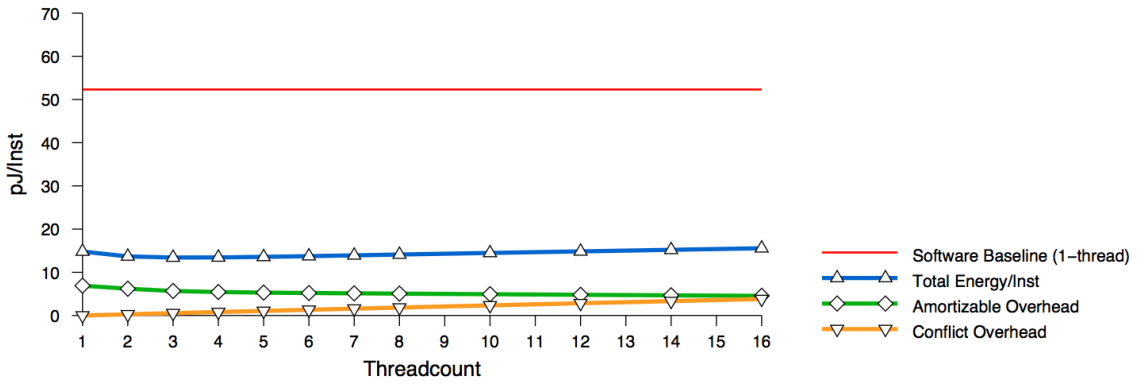


图 4-3 同时多线程的优势

图 4-3 展示了当负载和特定目标 CoDA 架构中专用协处理器分布匹配时，逐渐增加的同时运行的线程对平均每条指令消耗的能量的影响。当负载和专用协处理器匹配时，线程之间的竞争情况其实很少。图 4-3 中分别画出了每条指令执行时所消耗的平均能量，以及两个新增加的组成部分：1）分摊开销（例如，多个同时运行的线程可以分摊固定的漏电功耗开销）；2）冲突开销（例如，由于竞争某一个专用协处理器失败导致线程在通用处理器上执行而不是在 C-core 上执行所带来的额外能量开销）。为了方便观察，图 4-3 中最上面常数线条描述了系统中只有一个软件线程在主处理器上执行时平均每条指令所消耗的能量。

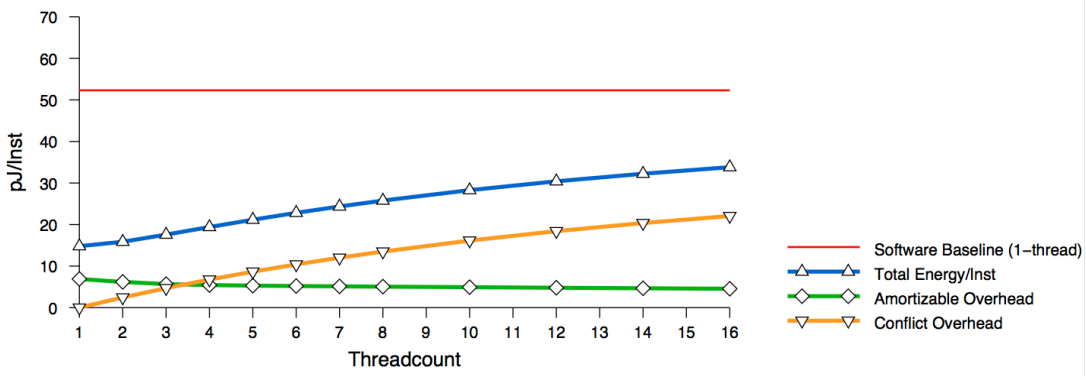


图 4-4 竞争冲突开销

数据表明增加 3 个同时运行的线程可以将平均每条指令运行的能量消耗降低 9%。增加的线程超过 4 个时，冲突消耗的能量超过均摊所减少的能量开销。当运行 16 个线程时，每条指令所消耗的能量与单线程相比增加了 5%。

与图 4-3 中的数据不同，图 4-4 虽然采用了相同 CoDA 设计，但是所运行的负载是非均匀分布的。这样会产生负载与专用协处理器分布的不匹配情况，也就产生较多的线程对专用协处理器的竞争冲突，并导致较低的能量效率。在这种情况下，随着线程数量

的增加，冲突开销导致了平均每条指令所消耗的能量快速增加。当运行 16 个线程时，每条指令所消耗的能量增加了 2 倍多。冲突开销增长的如此迅速是由于负载的分布与生成 CoDA 架构的训练数据集不匹配所导致的。

4.3.2 融合专用核与竞争缓解

在大部分情况下，生成 CoDA 架构时就对未来系统负载进行精确的 profile 是不可能的，所以系统运行时对专用协处理器的竞争冲突也是不可避免的。但是，架构师可以采取一些方式来降低这种影响。最简单的减少冲突开销的办法就是对专用协处理器进行复制来提供额外的硬件计算能力。但是这种方法几乎会增加两倍的 CoDA 芯片面积，并使单个线程的能量效率降低 23.4%。因为这会增加将近一倍的漏电功耗开销以及增加互连开销。

为了降低复制专用协处理器所带来的面积开销，架构师需要注意这个事实：在大多数情况下，应用程序是很少使用这些冗余的 C-core 的。这就促使架构师可以通过融合多个冗余的 C-core，并使用这个冗余的 C-core 来减少简单复制多个不同 C-core 所带来的影响。文献[19]中描述了如何融合 C-core。该研究中展示了如何自动地识别目标函数中与其他函数相似的部分，并生成一个融合的专用协处理器的过程。所生成的专用协处理器不但支持多个不同代码段，而且也仅仅比没有经过融合的 C-core 面积大一点点，能量效率低一点点。文献[19]的实验结果表明：融合后的 C-core 在覆盖更多软件功能的情况下，面积比单独的多个 C-core 减少 23%，同时与这些单独的 C-core 相比能量效率降低 27%。因为对于专用逻辑来说动态能量消耗仅仅是总能量消耗的一小部分，所以这个权衡通常可以为面向多线程负载的 CoDA 架构带来能量效率的优势。

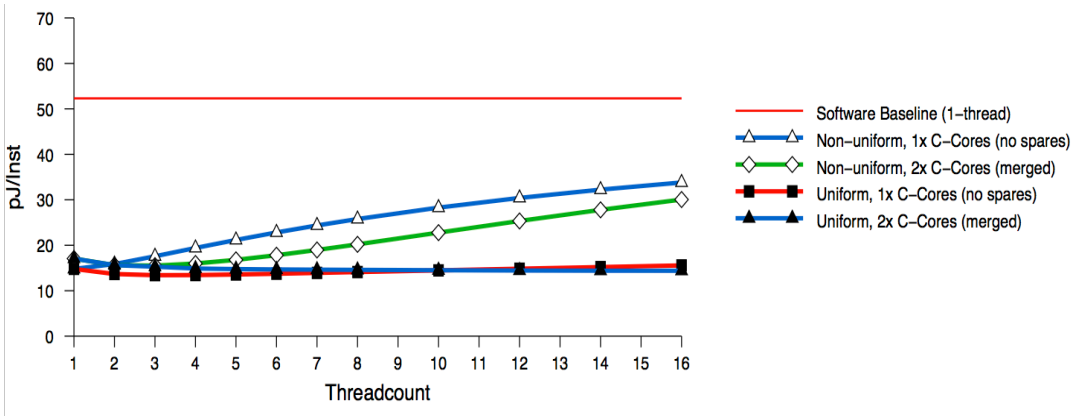


图 4-5 冗余 C-core 的好处

为了量化的分析融合所带来的好处，本节创建了一个新的 CoDA 架构。在新系统中使用融合冗余的方式为每种类型的 C-core 在功能上提供 2 倍的数量。这使新 CoDA 架构，花费了额外 41%的面积而且降低了 15%的能量效率（与单线程负载相比）。图 4-5 展示了与图 4-3 和图 4-4 比较，当系统中包含融合的 C-core 时平均每条指令的总能量消耗。融合技术不但可以为均匀负载分布（下方两条线）也可以为非均匀负载分布（中间

两条线）带来好处。对于均匀负载分布来说，在运行 16 个线程时包含融合 C-core 的系统可以提高 7.4%的能量效率。对于非均匀负载分布来说，因为负载与 C-core 不匹配，融合技术可以改善最多达 22.1%（运行 7 线程时）的能量效率，并且在运行 16 个线程时优化 11.1%的能量效率。

4.3.3 并发执行与访存带宽

增加 CoDA 架构中同时运行的线程的数量所引起的另一个值得关注的问题是对有效带宽的利用和竞争通信资源。然而当 C-core 数量急剧增加时，CoDA 架构所支持的最大并发执行线程限制在瓦片的数量。更进一步，表 4-1 中面向 128 个应用负载的 CoDA 设计中具有最优化单线程 EDP 的设计仅仅使用了 16 个瓦片，也就是这种情况下最多可以有 16 个线程并发执行。

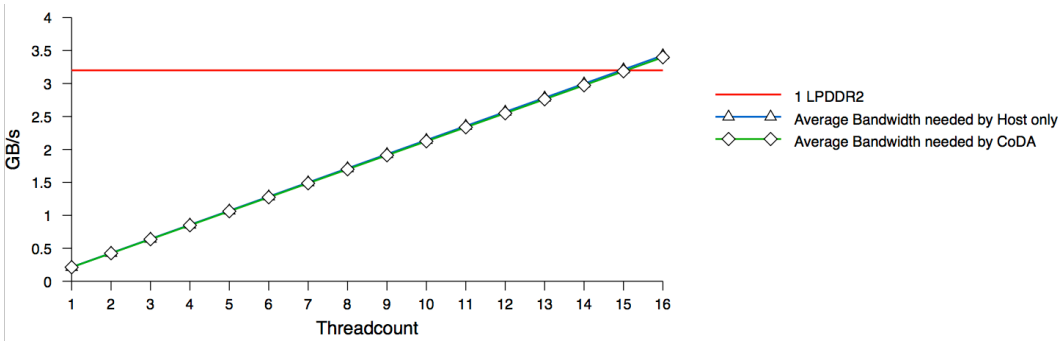


图 4-6 片外存储带宽使用情况

在瓦片内存储系统采用顺序发射以及阻塞执行的方式，当系统只有 16 个瓦片时同一时刻 CoDA 架构中向外围设备发起的最大存储器缺失数量为 16，并且部分缺失还可以在二级高速缓存中命中。MIT 的 Raw[72]处理器也具有 16 个瓦片，并且采用了与本研究相同的片上网络和相似的存储系统（尽管 Raw 处理器没有二级高速缓存）。在 Raw 处理器的 16 个瓦片上同时运行相互无关的 SPEC 测试程序所带来的性能损失小于 7%。在实际系统中，本文发现因为一级缓冲采用阻塞执行的方式，CoDA 架构与原始的通用处理器执行方式相比，并没有显著的增加一级高速缓存缺失后请求二级高速缓存的频率。本文评估了所有测试程序需要的片外存储器平均带宽，然后计算了当运行越来越多的线程时所需的片外存储器带宽，并假设这些测试程序在系统中均匀分布。如图 4-6 所示，本文的实验也表明，在计算并使用负载平均带宽的情况下，CoDA 架构平均对片外带宽的需求是十分普通的，仅仅使用一、两个 LPDDR2 通道就能满足。本文也认识到，选取不同的负载对带宽要求是不同的。例如运行 16 个 MCF 测试程序时，带宽竞争将成为影响性能的主要因素，这种负载比本文所考虑的负载需要 3 倍以上的带宽。这种情况可以在 2D-mesh 网络周边节点挂载更多地片外存储节点缓解。

4.4 优化 CoDA 能效的潜在方法

通常系统和芯片的设计者可以在以下 4 个不同层次上对能耗进行优化，当然有些方

法需要多个层次的技术支持。在工艺级上，由于本文的模型已经考虑了现在 Intel 的 3D 晶体管（FinFET）技术，所以进一步优化需要观察未来的技术发展。从 ITRS 最新公布的 2013 年路线图可以知道，未来在工艺级降低功耗和能耗的方法需要更进一步的研究，研究的主要方向有以下几个：部分耗尽绝缘体上硅（PDSOI）[85-89]、全耗尽绝缘体上硅（FDSOI）[90-93]、BULK 技术以及 Multi-Gate（MG）[94-96]等工艺。此外，由于 MOSFET 的亚阈传导漏流是器件的基本属性，所以除非工艺级有颠覆式的进展（例如：使用新型的器件取代 MOSFET），否则并不能从根本上解决暗硅问题。

在电路级上，可以尝试探索使用的技术包括堆栈效应电路[97, 98]、动态多阈值 CMOS（Dynamic Multi-threshold CMOS）电路[99]、双阈值电压电路（Double-Volage）[100-102]、在通信逻辑如片上网络中使用异步逻辑[103-108]等等方法。这些方法往往都是基于器件的物理特征，通过改变晶体管的微结构并用新的制造工艺以及更加强大的开发工具来实现。

最常见的研究是属于体系结构级的优化。对于 Cache 的动态功耗优化，大部分优化的目标是为了减少访存的次数或者减少每次访存所需的能耗。常见的减少访存次数的方法大部分是各种基于缓冲的技术[109-114]。这种技术的基本出发点就是在体系结构中增加一个更小的缓冲，用于存储更有可能被立即访问的数据，使得处理器的访存大部分被这个小缓冲过滤，以减少对 Cache 的访问。这些较小的缓冲由于占用逻辑少、功耗较低，所以使用这些技术可能进一步降低 CoDA 中 Cache 的动态功耗。减少每次访存所需能耗的方法也是多种多样的，有基于对访存行为进行分析并预测未来访存数据的路预测技术[115-118]；也有分步骤访问标签和数据的方法[119]，这样在知道访存未命中时将不再访问数据存储器。这些方法往往工作在处理器核内部，所以可以较为容易的添加进 CoDA 的瓦片中。

对 Cache 的静态功耗优化的思路通常基于关掉不需要使用的部分 Cache，这需要 Cache 在体系结构上有部分的重构能力。通常 Cache 的可重构能力包括以下几种：容量的重构[120]，通常修改 Cache 中的组数；关联度重构[121-124]；缓存行大小的重构[125]等。这些方法都可能配合系统级的优化方法来进一步减少 Cache 的静态功耗。

对互连网络动态功耗的优化思路大体上也有两个方向：1）减少数据在互连网络上传输的次数，这种方法可以使用更加智能的数据请求和应答机制甚至使得底层存储器可以较为有效地推送数据使得减少这种请求和应答次数，或者更加智能的任务映射机制[126-129]；2）尽量减少数据传输的翻转，这种方法往往使用一些编码技术[130]。

系统层级的优化往往是从整个系统的角度出发的，常见的方法包括动态电压/频率调节技术（DVFS）[131-133]、多时钟域技术[134, 135]、多电压域设计[136, 137]以及类似 Intel 的深度电源关断技术[46]。虽然本文的 CoDA 架构模型中已经对这些技术进行了建模，但是建模的粒度与主流商用处理器的实际情况还有一定差距。例如，本文的模型中仅仅有 4 个电压域，而东芝在 2011 年设计的 SoC 具有 25 个电压域，预计未来芯片将有更多的电压

域；对于时钟域，华为海思的麒麟 920 处理器包含了几百个时钟域。因此对于进一步研究 CoDA 架构设计，还需要探索更细粒度的系统级功耗管理方法，并进行更细致的建模来观察功耗的分布情况。

从图 4-2 中可以看出以下几点，1）对于 Cache 的静态功耗优化优先级要高于对 Cache 动态功耗的优化，因为在 CoDA 架构中 Cache 的静态功耗要远高于动态功耗。2）对于互连网络，由于动态功耗远高于静态功耗，所有对于互连网络的动态功耗优化优先级要高于静态功耗优化。3）进一步研究控制大量专用协处理器的静态功耗也十分重要。

对于优化 Cache 的静态功耗前面所提到的 Cache 重构以及系统层级的方法将较为有效。对于减少互连网络的动态功耗在前面提到的方法外，提出以下额外的潜在方法。

预取（Prefetching）[70, 138-140]是常见的提高 Cache 命中率并提升系统性能的方法。但是预取也会造成大量额外的无用数据传输，预取回的数据如果直接进入 Cache 还可能造成 Cache 污染以及数据颠簸问题。而从低层次存储器向高层次存储器推送（Push）数据则正好相反，因为推送的过程不需要处理器核发起请求，并且推送来的数据较为容易判断是推送来的，这样就可以在高层次 Cache 旁边设计小缓冲来缓冲这些数据，防止对 Cache 中原有数据造成影响[141, 142]。

作者曾经对单核处理器中的推送技术进行了详细的研究，建模后使用时钟精确的仿真器进行了详细的仿真，发现仅仅使用简单的连续推送技术（对指令进行增量连续推送；对于数据由于程序使用的数据中存在栈所以进行地址增和减两个方向的推送），就可以使得处理器对二级 Cache 的访问减少 34.6%，有效减少了请求数据包在互连上的传输次数。一级指令 Cache 的缺失率减少 74.6%，一级数据 Cache 的缺失率减少 39.6%。整体上程序的 IPC 可以提高 6.6%。这个推送模块的资源消耗与高速缓存相比仅占 0.2%。更进一步，如果结合 L0 Cache 技术，将更加有效的提高访存在小缓冲中的命中率，这样对降低 Cache 的动态功耗也是有利的。此外，IPC 的提高说明程序运行速度加快，运行时间将缩短，这也有助于降低整个程序运行的能量消耗。这个研究发表于论文“Tolerating memory latency: L2 cache actively push architecture”和“AAP and AAPM: improved prefetching structures of the L2 cache”。

在 2D-mesh 互连的 CoDA 架构中，L1 距离 L2 更远，所以减少的通信将更为有意义。但是在这种情况下，如何维护缓存的一致性以及推送带来的通信开销等问题也需要更进一步的细致研究。

此外有一些“性能功耗比”较高的技术也是值得探索的，例如同时多线程技术就由于“性能功耗比”较高而应用在 Intel 的超低功耗移动处理器中[46]。另外作者针对较小的加法器电路尝试使用过混合工艺库流片的技术，这种技术的基本思想是电路的关键路径使用 HP 库中的标准单元，而在非关键路径使用 LP 库中的标准单元，单元的替换在后端流程中使用 PrimeTime 进行。实验表明这种方法效果较好，但是目前的自动替换程

序工作还是较为缓慢，无法应用在大型电路上，有必要对替换算法进行进一步探索。

4.5 相关研究

随着暗硅现象的加剧，架构师正在向通用架构中集成越来越多的专用协处理器。GPU 是一个特别典型的例子，英特尔和 AMD 最新提供的芯片都将 GPU 和处理器集成在一块芯片上。最近很多人[143-145]也在尝试扩展编程语言来驾驭这些新出现的异构计算平台，例如为 GPU 所设计的 CUDA[146]，为流框架所设计的 Brook 语言[64]，但是他们所关注的都是具有高度并行和松散耦合的程序执行模型。

即使是较为灵活的异构处理器架构，例如英特尔的 EXOCHI[144]，也面临着在同一个设计中如何使用 1000 个不同专用协处理器这样的挑战。EXOCHI 中将顺序执行程序映射到异构执行引擎需要为每一块目标硬件提供特定的编译器。

以前人们研究过使应用程序的大部分在基于可重构的硬件上执行；与之不同，本文的设计中应用程序大部分运行在特定的专用协处理器上。例如，Tartan[147]将整个程序映射到层次化的粗粒度异步可重构结构上。可重构逻辑具有较好地灵活性，但是 Mishra[54]评估发现如果要映射整个程序那么就需要硬件的虚拟化技术，这将大大增加性能和能量开销。

因为可以减少线延长，瓦片化成为一种提高系统扩展性的通用方法，例如 MIT Raw 处理器[72]，UT Austin 的 TRIPS 处理器[148, 149]以及华盛顿大学的 WaveScalar 处理器[150]。由于瓦片结构可扩展，所以可扩展的 CoDA 架构也采用瓦片结构。采用瓦片结构也使得大量的协处理器分散在多个存储器以及主处理器接口之间。GreenDroid[3, 4, 6]研究论文中提出了在使用协处理器的系统中使用瓦片架构，但是 GreenDroid 的研究并未涉及本文所解决的系统可扩展性问题。

Hannig[151]描述了一种通过侵入性计算模式动态地将计算映射到异构多核 SoC 的模型。而 CoDA 架构也可以潜在地从这种探索并行资源中获得好处，现阶段的 CoDA 研究集中关注于降低主要的串行应用所消耗的能量。此外，本文有意地使 CoDA 设计可以运行未经修改的传统软件的源代码，仅仅需要编译器在程序的函数与硬件所覆盖的函数之间做映射，这就使得 CoDA 设计对于程序员是透明的。为 CoDA 架构设计新程序的最有效方法，是一个值得进一步研究的新课题。

与 CoDA 类似，文献[152]也提出了一种面向暗硅时代而设计多核系统的方法。该研究主要面向体系结构相同，而每个核针对不同应用进行不同电压和频率优化的架构，而且仅仅讨论了运算核的能量效率。与之不同，CoDA 的研究粒度更细，并且讨论了更加异构化的运算器件。另外本章不仅讨论了运算核，而且也讨论了存储系统和互连结构。

前人的研究，例如文献[54]，主要侧重于研究设备虚拟化和管理加速器内部的的局部存储器来探索多线程和协处理器之间的交互关系。相比之下，CoDA 架构中使用的 C-core 本来就具有一致性，所有不需要内部具有较大的私有存储器。C-core 也可以通过融合[19]

来缓解资源的冲突。这种融合方法可以既不增加 C-core 数量，也不增加对虚拟化的支持。

文献[5, 13]探索了如何使程序的大部分运行在基于门控时钟的协处理器上，但是这些协处理器并没有使用门控电源。图 4-1 表明， 由于 CoDA 架构中在大部分时间中大部分的硅都将处于闲置和关断电源的状态，所以门控电源是影响较重负载下大规模 CoDA 架构的能量效率的关键因素。设计人员需要使 CoDA 架构在默认情况下从一开始，所有的处理器部件都处于深度休眠的状态。这种运行模式在微尘传感器[153]和其他能量非常关键的系统中非常常见，但是他不是通用处理器的传统运行模式。

4.6 小结

本章研究了将成百上千的专用协处理器集成到通用架构所带来的可扩展性方面的问题。主要关注点在于，面向大规模的应用负载 CoDA 架构是否还可以成倍地提高能耗效率。经过系统性地研究 CoDA 的设计空间表明，可覆盖 128 个应用的可扩展的 CoDA 设计，可以为整个负载带来 3.7 倍的能量优化和 3.5 倍的 EDP 优化。这充分说明了 CoDA 架构的能量效率优势，进而证明了 CoDA 架构具有能量扩展性。本章还发现在大规模瓦片结构设计的可扩展 CoDA 架构中，制约能量效率的关键因素是暗硅部分的漏电功耗、互连的开销以及存储系统。

实验结果表明线程竞争共享的专用协处理器限制了能量效率的提高，但是 CoDA 设计可以提供一种高效的“冗余专用协处理器”的方法来解决这个问。在线程平均分摊漏电功耗，互连和存储系统开销的情况下，使用该方法使得运行多线程负载的 CoDA 与单线程负载相比可以获得 3.8 倍的平均每条指令能量优化。

本章的主要内容发表于论文“Exploring Energy Scalability in Coprocessor-Dominated Architectures for Dark Silicon”。4.4 小节部分内容发表于论文“Tolerating memory latency: L2 cache actively push architecture”和“AAP and AAPM: improved prefetching structures of the L2 cache”。

5 CoDA 原型系统设计与实现

在前文详细描述了 CoDA 架构、C-core 的结构、S-core 结构的基础上，本章着重关注和探索硬件实现 CoDA 架构的问题。硬件实现的 CoDA 架构不但可以验证设计思想的可实现性，并且也可以验证设计的正确性尤其是自动生成的 C-core 以及 C-core 与主处理器交互的正确性。本文的实现分为两部分：第一部分使用 FPGA 搭建了原型系统，第二部分探索了简单 CoDA 的 ASIC 芯片实现。本章主要内容如下：

第一小节介绍 Basejump 快速原型搭建系统，该系统是搭建 CoDA FPGA 原型系统的基础。该系统 FPGA 设计实现、软件驱动、操作系统修改部分由本文作者开发。系统已经开源给加州大学圣克鲁斯分校和康奈尔大学，并且也被业界的 MaXentric 公司使用。简单来讲，Basejump 提供了一整套用户定制芯片、FPGA 开发板以及主机的互连接口，多种外设控制器以及 Linux 上的驱动程序。开发团队使用 Basejump 可以快速高效的建立原型验证平台和芯片验证平台。此外，Basejump 通过采用跨芯片的环形网络可以使得 2D-mesh 网络跨芯片逻辑扩展，这样设计师可以使用多块 FPGA 一起来验证较大的 ASIC，或者使用 ASIC 和 FPGA 一起来验证未来版本的 ASIC 芯片。

第二小节介绍将简单 CoDA 架构 GreenDroid 集成到 Basejump 架构后所搭建的完整 FPGA 验证平台，以及如何测试验证所有电路的正确性。其中包括单独验证每一个模块的模块验证以及验证整个系统和 C-core 的测试程序。

第三小节探索简单 CoDA 芯片的流片工作。目标芯片包含两个瓦片：一个瓦片包含主处理器和一个 C-core；另一个瓦片为第二章介绍的 S-core，这是集成所有本文提供的专用协处理器的最简系统。本章完成了芯片的前端设计、IO 设计、以及后端设计。

5.1 Basejump 快速原型开发系统

当设计师开发新系统和设计芯片的时候，往往需要集成各种各样的外设控制器以便通过这些外设连接其他设备，共同来支撑目标系统的正常运行，例如对开发的处理器进行测试，就至少需要集成主存储器。集成外设控制器不但要消耗大量的工作时间，并且外设控制器的调试往往也较为困难。基于以上原因以及实际项目开发需要，本文开发了 Basejump 系统。该系统可以帮助设计人员快速建立原型系统，并节省大量的外设和驱动开发时间，使研发团队可以更专注于自己的设计。另外，在 CoDA 架构实际开发过程中，发现单块高端 FPGA 资源无法满足验证 ASIC 芯片的需求。所以针对 2D-mesh 互连的众核架构提出了由跨芯片环形网络连接的跨芯片逻辑可扩展 2D-mesh 片上网络，并为跨芯片的每一个 2D-mesh 物理通道分别提供跨芯片的流控机制。

Basejump 的基本思想见图 5-1，红色虚方框里面的就是 Basejump 系统设想图。其中 Northbridge 用于芯片与芯片间以及开发板与开发板之间的互连，Plug Board 用于连接

各种不同类型的外设。这样芯片开发团队就可以简单地在设计的顶层添加适配电路（接口电路）并将设计连接到 Northbridge 上，这样就获得了一系列的外设支持，并搭建了较为完整的原型验证平台。图 5-2 是本文最终所设计的整个系统，其中红色虚线里面是 Basejump 系统。该系统大体上分为三大模块：子板（可以是 FPGA 开发板也可以是 PCB 版）、母板（FPGA 开发板）以及上位机（Linux 主机）。蓝色虚线框出了三大模块之间的边界。

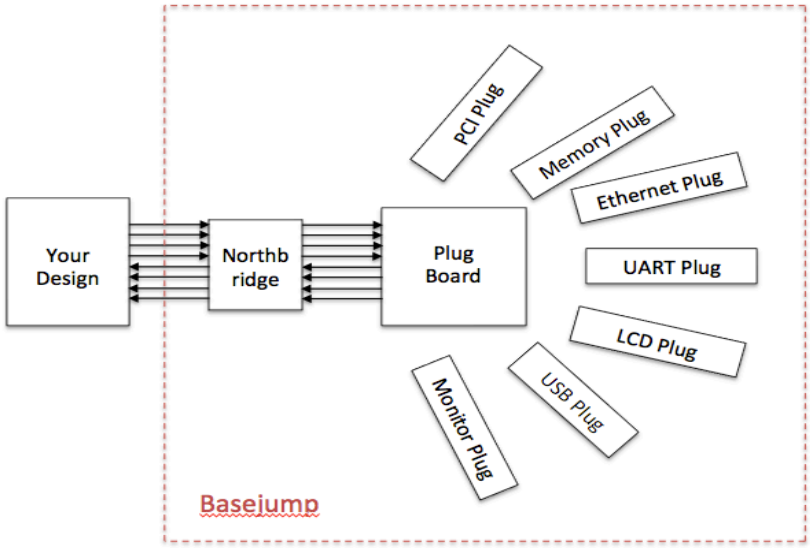


图 5-1 Basejump 概念图

其中子板上的逻辑分散在两块芯片中：其中北桥（Northbridge）是一块 FPGA 芯片，该 FPGA 负责子板上芯片之间通信以及子板与母版通信，也就是说既有芯片间通信功能也集成了开发板间通信逻辑。子板与母版通信采用 FMC 接口，通过这个接口实现了 LVDS 高速通信；芯片之间通信采用 MURN 网络（Multi-University Research Network）进行通讯，该网络提供了 4 个通信通道，每一个通道都可以单独通信也可以多个通道绑定（Bonding）在一起共同工作。这样设计的目的是避免芯片加工时由于管脚出错而导致整个芯片都无法与外界通信，并无法使用的现象发生。子板上另外一块芯片可以是流片后的用户定制 ASIC 也可以是用于验证的 FPGA 芯片。团队其他成员为这 2 种情况分别设计了 PCB 板。Basejump 还提供了一组跨芯片环形网络，用户设计集成 Basejump 时仅仅需要为设计添加环形网络适配器连接到环形网络上。环形网络的主要作用有以下几点：1）支持多个用户设计一起流片，这样不但可以分担流片成本，而且也可以分担 IO，PCB 等等的设计工作。2)通过跨芯片环形网络连接子板和母板中的 2D-mesh 片上网络，使得 2D-mesh 在逻辑上跨芯片扩展，这样就可以使用多块 FPGA 来共同验证大规模的 ASIC 设计。3）可以使用环形网络向系统中集成其他多种外设。

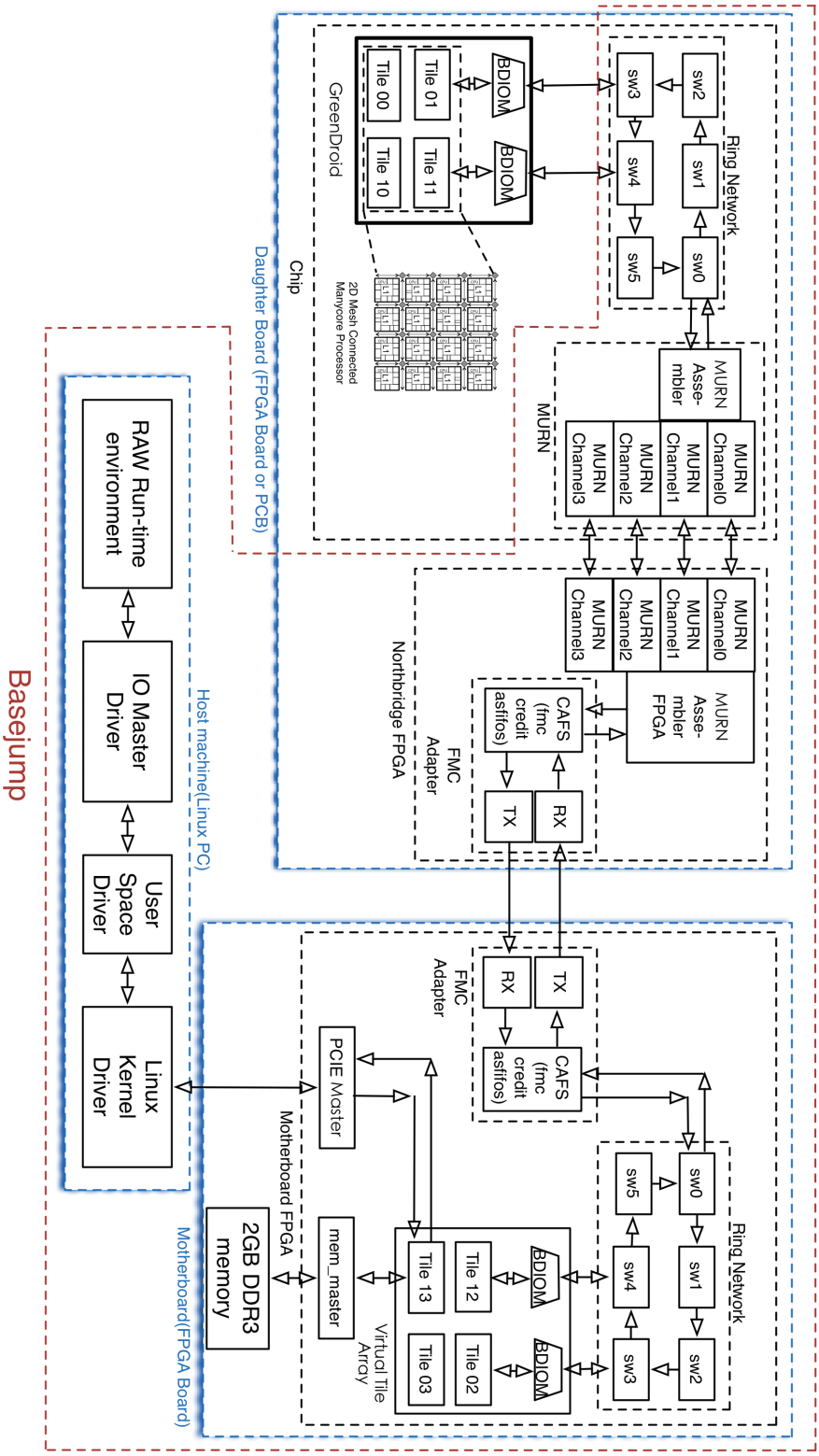


图 5-2 系统模块图

母板是一块 FPGA 开发板，其主要的作用是集成各种外设控制器并通过接口连接外设，同时可以通过 PCIE 接口连接上位机，并且当子板 FPGA 资源不足时可以分担部分 ASIC 逻辑。目前可以提供的调试成功的外设控制器有 DDR3 内存控制器、Ethernet 网络接口、串行通信接口、FMC 控制器、PCIE 控制器以及 LCD 控制器。母板的 FPGA 中

也提供了环形网络和瓦片阵列以对应芯片中的逻辑。瓦片阵列可以是真实地阵列（包含实际的处理器或用户设计），也可以是虚拟的阵列（仅仅包含 2D mesh 互连网络）。当单块 FPGA 芯片资源无法验证 ASIC 逻辑的时候，可以在母板上集成真实地阵列，并在阵列的处理器上集成专用处理器。虚拟阵列的作用仅仅相当于 Plug Board，可以连接多种外设。当用户不需要连接多种外设或者用户的设计不使用 2D mesh 片上网络时，可以设计适配器并直接将 FMC 控制器与需要的逻辑相连，或者将外设直接连接到环形网络上。

通常上位机是一台 PC 电脑或者服务器，作者为上位机提供了 Basejump 系统的驱动程序（包括系统内核驱动和用户空间的驱动）、IO 通信通道的控制驱动和系统运行时环境。其中 Basejump 驱动程序实际上就是 PCIE 的驱动程序；IO 通信通道可以对数据传入和传出 2D mesh 网络做流量控制，保证不丢包；系统运行时环境可以帮助模拟操作系统，方法是模拟执行一系列的系统调用并将这些系统调用的结果返回给用户设计。这样就简化了用户开发处理器时的验证环境搭建的复杂性。本设计面向的上位机搭载的是 Linux 操作系统。

由于 Basejump 中包含多种多样的互连结构，例如片上的 2D-mesh 网络、片上的环形网络，芯片间的并行互连和串行互连等等，较为复杂。所以在大体上了解 Basejump 的模块组成之后，本章来详细介绍 Basejump 一些设计要点，首先介绍各个通信部件之间的流控，其次介绍新开发的一些重要组件的协议，再次介绍跨芯片扩展的 2D-mesh 网络以及其他重要内容。

5.1.1 Basejump 中的流控设计

Basejump 中所有临近模块以及跨芯片、跨 PCB 板的通信都进行了专门的流控设计（flow control），这样既可以保证通信的速度又可以保证通信的准确性避免数据包丢失。流控的基本思想是数据的发送方在发送数据时确切地知道数据接收方是否有空间可以准确的接收数据，当接收方有空间时连续发送数据直到检测到接收方无法接收更多的数据，此时发送方将进入等待的状态直到接收方可以继续接收数据。这样相对于之前使用的握手方式将大大减少等待时间，提高通信的效率。

流控的关键在于确切知道接收方是否有空间可以接收数据，通常情况下流控设计中的接收方会在接收端添加 FIFO，而发送端添加流控逻辑计算接收方剩余的 FIFO 空间。剩余空间的计算基于以下 3 个信息：1）接收方 FIFO 的初始大小；2）发送方新发送的数据需要占用的空间；3）接收方的 FIFO 新释放出来的空间。其中第一个和第二个信息通常可以在发送端直接获得，如何获得第三个信息成为设计要点。

根据流控下的通信模块的位置距离关系，Basejump 中的流控大致分为 3 种方式：1）临近模块间。2）临近芯片或 PCB 板间。3）非临近的跨芯片逻辑。

对于临近模块可以在接收方设计专门的信号线负责通知发送方有额外的 FIFO 空间

被释放，例如 Basejump 中与 2D mesh 网络互连模块，环形网络各个交叉开关之间的互连。对于临近芯片或 PCB 板的设计，可以使用专门的或者复用的 IO 通知发送方有额外的 FIFO 空间，例如 Basejump 中 MURN 网络的通道和 FMC 适配器。非临近的跨芯片逻辑之间的流控方法更为灵活复杂，因为他们之间传输的数据往往需要打包和拆包，并且这些数据要经过很多模块进行传输甚至经过多次协议转换，专用的信号线可能会占用宝贵的带宽，因此通常使用的方法是将释放空间的信息与数据一起打包进数据包或者为空间释放信息增加单独的虚通道。如果采用虚通道的方式，通常设计会设定一个阈值，当释放的空间数量超过阈值时生成一个数据包并传送给发送方。在 Basejump 中，跨芯片可扩展 2D-mesh 中使用的 BDIOM 之间的流控就是这种非临近的跨芯片方式。

本设计所使用的 2D mesh 网络协议和流控方式与 MIT Raw 处理器相同。这里就仅对接口进行介绍。接口和流控模块如图 5-3 所示，其中发送方连续发送数据（data，32 位）和数据有效信号（valid），直到发现接收方没有剩余的 FIFO 空间。接收方监控 FIFO，当有数据从 FIFO 取出则通知发送放 FIFO 又增添一个剩余空间（通过 thanks 信号）。

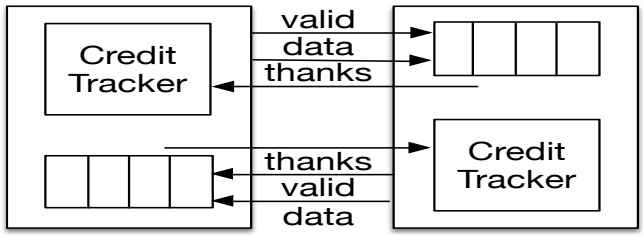


图 5-3 2D mesh 网络接口和流控模块

其余两种流控方式分别在介绍 MURN 协议和 BDIOM 时进行介绍。

5.1.2 芯片间通信 MURN I/O 协议

MURN I/O 协议是一套芯片外和芯片内通信接口标准，这套标准支持基于多个通道的通信方式。从芯片外进入芯片的数据包必须满足 MURN I/O 协议，一旦数据进入芯片并经过 MURN 模块后需要满足环形网络协议。本小节详细介绍 MURN I/O 接口协议，硬件逻辑模块以及启动时的校准工作，下一小节介绍环形网络协议。

因为 MURN I/O 的目标使用情况是连接用户定制 ASIC 芯片和 FPGA，那么在用户定制芯片的封装管脚限制和 Die I/O pads 的数量限制以及流片成本的制约下，MURN I/O 的逻辑信号都采用单端型（single-ended 对应于 LVDS 这种成对信号）信号。在 MURN I/O 协议的设计阶段，考虑到在芯片加工时可能会出现 I/O 管脚加工问题导致某些管脚失效的情况，所以针对这种情况本设计采用多个通道并且可以仅仅依靠至少一个有效通道进行通信。另外考虑到各个通道的金属线在 PCB 板上的走线长度可能不一样，本设计也保证了各个有效通道可以工作在不同频率和相位的时钟下。

最终本设计的 MURN I/O 包括 4 个通道，支持 1 个通道独立工作、任意两个通道绑定工作以及 4 个通道绑定同时工作。在绑定情况下，各个通道也是可以工作在不同的时

钟下的。时钟由 North Bridge 提供，并且在系统校准的时候确定可以使用的通道和每个通道的工作时钟。每一个通道有双向的通路组成，一个通路的数据位宽为 8 位。由于通道内部的数据还是并行发送，所以没有必要做数据的位校准（bit alignment）。本设计的 FMC 适配器中使用的 LVDS 采用串行通信，所以需要使用数据位校准。

图 5-4 是 MURN I/O 的结构框图，左侧展示了 4 个 MURN I/O 通道，每一个通道数据宽度是 1 字节并且可以独立使用。当系统启动时通道要进行校准，校准的目的是确定通道是否可用以及通道在什么时钟频率下能正常工作。当校准完成后，通道发送 enabled 信号给通道绑定控制器（murnio assembler cbc）来确定哪些通道可以绑定在一起工作，并将生成的绑定控制信息发送给输出拆包和输入打包逻辑。这样拆包（murnio assembler out）逻辑就会将环网传输过来的 80 位数据按照可用通道情况进行拆包并发送到片外，打包逻辑（murnio assembler in）也会将片外传入的数据正确打包成环网数据传入环网。由于芯片的 IO 工作频率和芯片内部逻辑工作频率不同，所以每一个 MURN I/O 通道中包含两个异步缓存。其中一个是数据发送缓存（send fifo）用于发送数据到片外，另一个是数据接收缓存（recv fifo）用于接收片外数据。

每一个 MURN I/O 通道都包含一对反向的数据通路，每一个通路对芯片外都包含以下逻辑信号：8 位的数据信号、1 位的时钟信号、1 位的 Token 信号和 1 位宽的 Command 信号。其中时钟和数据是源同步的，Token 是用于流控的信号，Command 信号是标识当前传输的数据是校验数据还是程序运行所传输的数据。有某些特殊的校验数据用于通知 MURN I/O 的另外一侧通道准备好等。这样每一个通道单个方向有 11 位信号，双向需要 22 位；4 个通道总共需要使用 88 个逻辑信号。由于芯片管脚上的信号管脚翻转一次消耗能量较多（相对于内部逻辑），所以本设计的 Token 信号每翻转一次（0->1 或 1->0）表明流控的接收方 FIFO 释放了 2 个空间。

MURN I/O 的校准过程如下：

1）在复位信号失效后，北桥上的 MURN 等待 128 个周期，以待校准的两端电路都已经有效复位并准备好校准过程。

2）北桥端的 MURN I/O 中的每一个通道发送 128 字节预先设定的常数序列，这个常数序列由 16 个不同数据并重复 8 次产生。这些序列是经由通道中的数据发送缓存发送的。

3）在用户设计端的 MURN I/O 使用数据接收缓存接收这些数据并用数据发送缓存将这些数据传回北桥。

4）北桥上的 MURN I/O 检查接收数据缓存读出的数据。如果数据与发送的 128 字节数据匹配，那么这个通道就校准了。如果没有匹配，降低时钟主频然后跳转到第 2 步重新发送 128 个数据。设计设定了一个时间阈值，当某一个通道校准后开始计时直到计时器达到阈值，此时没有校准的通道视为校准失败。

如果所有通道都校准失败，则系统将一直挂起（pending），主机可以通过读取 Basejump 提供的特殊寄存器了解系统的状态。如果有至少一个通道校准，那么系统可以继续其他的启动步骤。

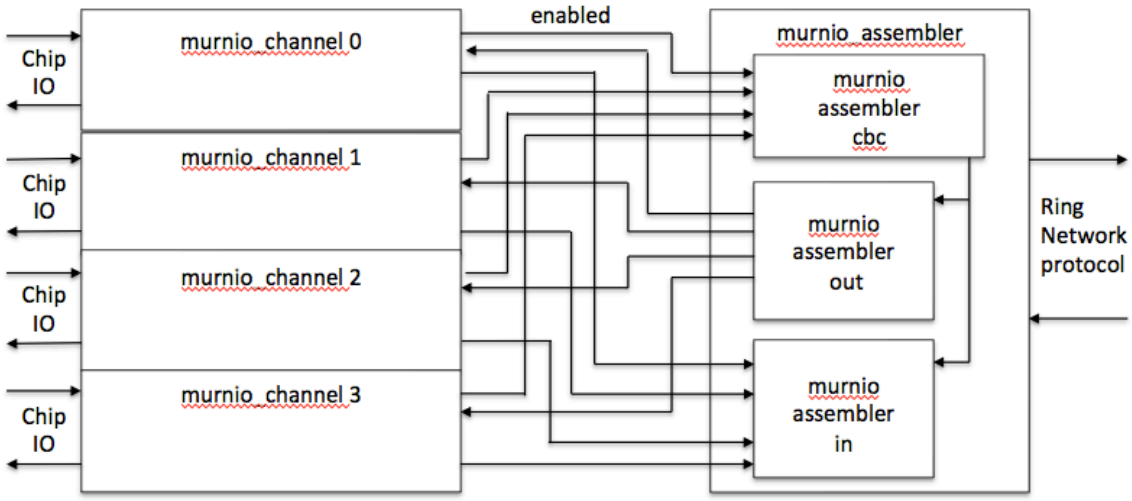


图 5-4 MURN IO 模块图

5.1.3 环形网络

环形网络在 Basejump 中扮演着重要的角色，通过环形网络系统可以连接多个不同的用户设计、多个芯片内的其他网络（例如 2D-mesh）、外设以及其他用于配置或者调试的模块。逻辑上环形网络在三块芯片间扩展。环形网络提供了一组由互连线或者芯片 Pin 连接的交叉开关，每一个交叉开关都可以连接一个网络节点。系统中环形网络的逻辑图见图 5-5。

在用户定制芯片上环形网络可以连接多个不同的设计，并且可以使这些个设计被单独测试；北桥芯片上的环形网络可以用于连接一些配置或者调试模块，例如本设计的配置网络控制模块，可以控制北桥芯片发送命令对用户定制芯片的某些 IP 或模块进行配置。在母板上面的环形网络通常可以镜像的连接瓦片阵列并连接多个外设。

环形网络中的交叉开关分为两类：1）监听模式开关，这类开关不判断所传输的包的目标地址而是直接将包传送给所连接的节点（或者直接将节点传来的包注入环形网络）。这类交叉开关通常连接的是用于通信的逻辑节点，例如图 5-5 中的 n0 和 n8。2）另一类交叉开关就是普通交叉开关，这类交叉开关解析数据包的目标地址并按照目标地址确定转发的方向。目前本设计实现的环形网络为单向环，下一次流片前准备实现双向环。交叉开关的类型可以通过参数进行配置，这种配置需要在编写 RTL 代码时完成。

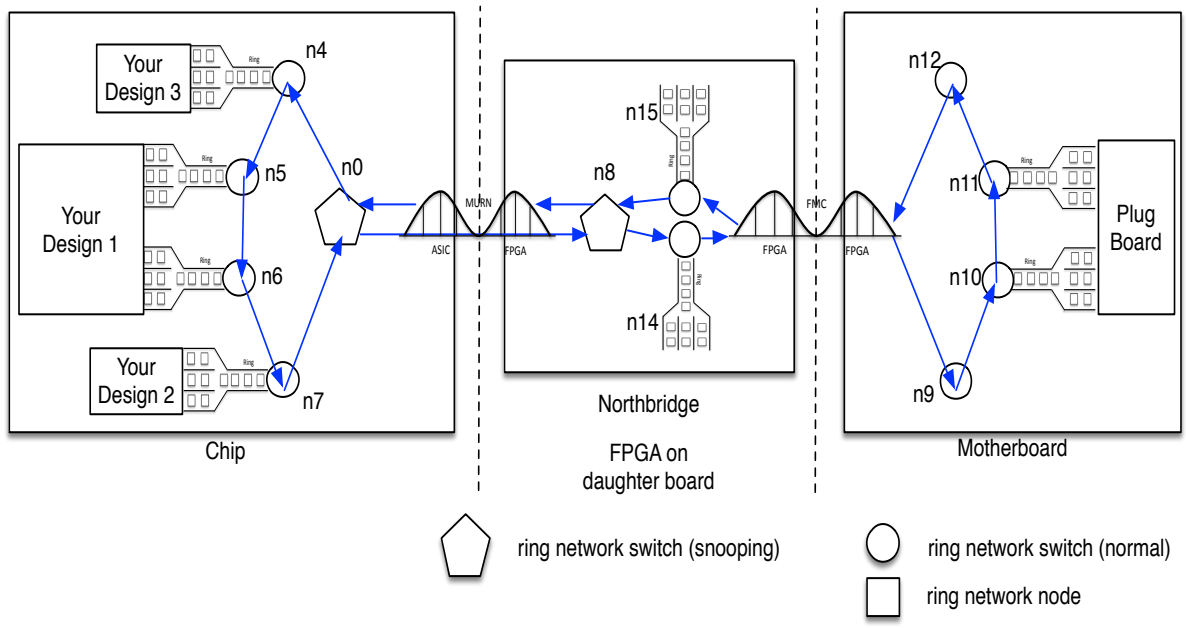
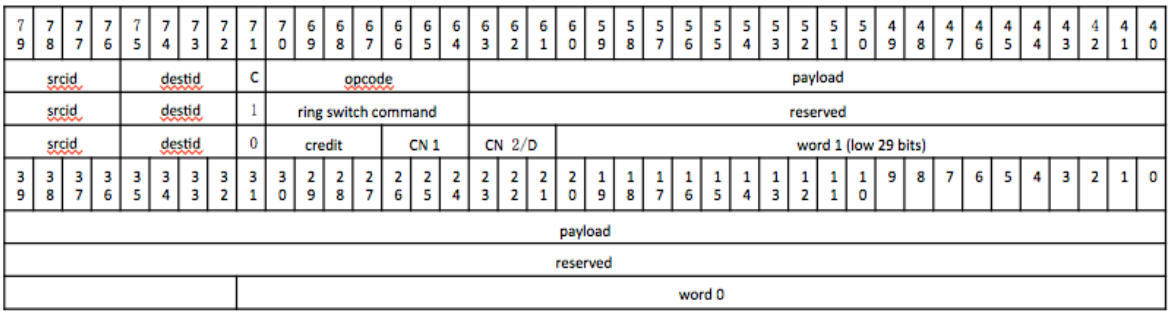


图 5-5 环形网络逻辑图

环形网络的数据包格式见图 5-6。环形网络的数据包为 80 位宽。其中前 4 位为源地址（source ID），用于标识产生数据包的节点。接下来的 4 位为目标地址（destination ID）用于标识接收数据的节点。后面的 1 位是命令位，如果这位为 1 说明这个数据包是给交叉开关的命令包；否则是给节点的数据包（在交叉开关看来这是节点的数据包，节点自己也可以使用这种包传输节点内部定义的命令）。接下来的 7 位为操作码位，当命令位为 1 时这 7 位就是给网络节点的命令，当命令位为 0 时这 7 位将由关联节点进行定义解析。后面的 64 位就是数据负载。

为了适应 32 位 2D mesh 网络，本文对节点数据包进行了特殊的设计。参见图 5-6 中环网数据包格式的第三行。当命令位为 0 时，接下来的 7 位操作码的前 4 位用于流控信息，来返回接收方 FIFO 释放出的额外空间。当环形网络和 2D mesh 网络做协议转换时，每一个环网节点最多支持 4 个 2D mesh 通道，并对每一个通道单独进行流控，每个通道占用一位的流控信息位，可以通过配置设定流控信息位为 1 时返回的信用（credit）数量。32 位数据到 64 位数据的转换如果仅仅简单的高位补零将造成 1 倍的带宽浪费，所以本设计的思路大体上是将 2 个 32 位 2D mesh 数据包打包进一个环形网络数据包。设计在电路中设置了时间阈值，当接收到一个 2D mesh 数据包后等待一定时钟周期，如果接收到第二个数据包则立刻打包成环形网络数据包。如果没有接收到第二个数据包，则将一个 32 位数据单独发送。这样的设计体现在了环网数据包格式上。7 位操作码的后 3 位不为 111 时，表明后面的 64 位都为有效数据，并且这三位标识了数据来自于哪个 2D mesh 通道。当这三位是 111 时，表明后面仅有 32 位有效数据（word 0），并且 word 1 的高 3 位标识 2D mesh 通道。



Command Name	Value	Description
RNDISABLE_CMD	1	Disable the Node connected to this switch
RNENABLE_CMD	2	Enable the Node connected to this switch
RNDOWN_CMD	3	Power down the Node connected to this switch
RNUP_CMD	4	Power up the Node connected to this switch
RNRESET_ENABLE_CMD	5	Enable the reset signal of the Node connected to this switch
RNRESET_DISABLE_CMD	6	Disable the reset signal of the Node connected to this switch

图 5-6 环形网络数据包格式和交叉开关命令

此外由于本设计采用的 2D mesh 网络建立通路采用虫洞的方式，所以从环形网络取出数据注入 2D mesh 网络时，需要将整个 2D mesh 网络消息都搜集全后一次性注入 2D mesh 网络。这样可以减少 2D mesh 中的网络等待时间。

目前本设计实现的交叉开关支持 6 个不同的命令，命令列表见图 5-6 的下半部分。这些命令主要控制关联节点的电源、复位信号和是否激活关联节点。其中 RNDISABLE_CMD 命令用于使相连节点失效，此时节点产生的所有数据包都不能注入环形网络，环形网络的数据包也不能进入节点。RNENABLE_CMD 命令激活相连节点，此时节点可以与环形网络通信。RNDOWN_CMD 命令关断相连节点电源。RNUP_CMD 打开相连节点电源。RNRESET_ENABLE_CMD 用于使相连节点的复位信号置位。RNRESET_DISABLE_CMD 用于使相连节点的复位信号失效。系统启动时这些节点的默认状态是关断电源（power off）并且非激活的。启动后打开某一节点的顺序为：1）发送 RNUP_CMD 命令打开该节点电源；2）发送 RNRESET_ENABLE_CMD 命令使该节点所有电路复位；3）发送 RNRESET_DISABLE_CMD 命令使该节点复位信号失效；4）发送 RNENABLE_CMD 命令使该节点激活。先复位再激活是因为这样可以防止激活后由于电路没有正确复位而向环形网络发送大量无用数据，而阻塞环形网络。

由于默认情况下所有节点都未上电，并且也都没有激活，这样就保证了虽然多个设计共同流片但是设计团队可以单独测试自己的设计。方法就是仅仅上电激活自己的设计。

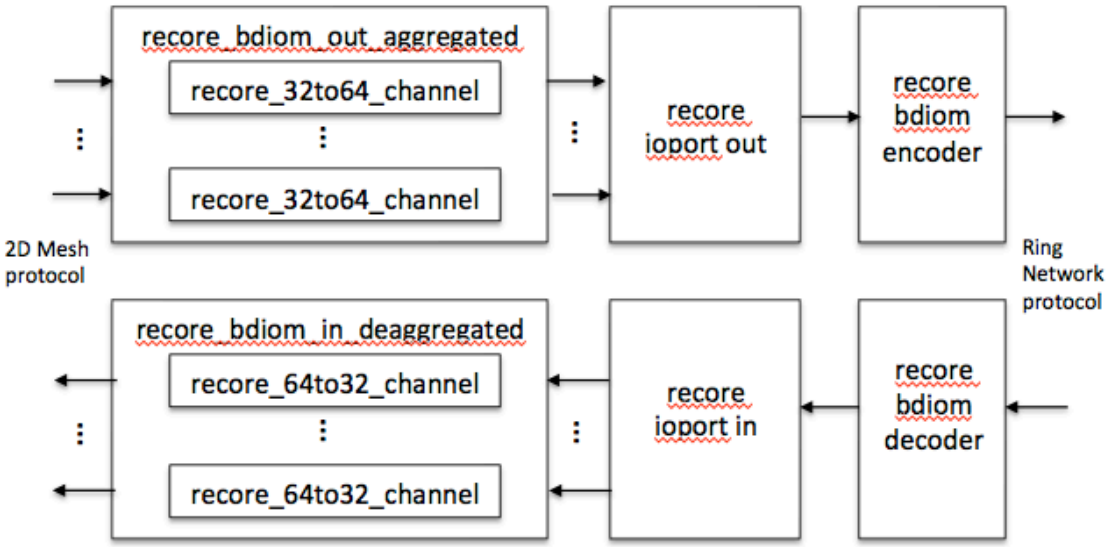


图 5-7 2D mesh 与环网协议转换器 BDIOM

图 5-7 是 2D mesh 与环形网络协议转换器 BDIOM 的模块图。目前每一个 BDIOM 最多支持 4 个 2D mesh 通道。由 2D mesh 向环形网络发送的数据，在 recore_bdiom_out_aggregated 模块就完成了 32 位到 64 位的转换；recore_ioport_out 模块的作用有 2 个：1）轮询（round-robin）挑选一个有数据要发送的 2D mesh 通道。2）查看这个通道的流控信息，确定接收方是否可以接收数据，如果可以接收数据就发送数据；如果不能就轮询下一个通道。recore_bdiom_encoder 负责数据打包成环网协议数据。从环形网络传入的数据经过拆包（recore_bdiom_decoder）后，确定数据属于哪一个 2D mesh 通道（recore_ioport_in），并将数据插入到该通道接收数据的队列中。负责搜集整个 2D mesh 消息并一次性注入 2D mesh 网络的逻辑在 recore_bdiom_in_deaggregated 模块中。

5.1.4 跨芯片扩展的 2D-mesh

由于 CoDA 架构芯片逻辑规模较大，单块 FPGA 芯片无法满足验证和建立原型系统的资源需求，所以本文尝试使用多块 FPGA 芯片来共同组建验证环境。此外，本文的 CoDA 架构目前采用了 2D-mesh 作为片上互联，所以本文提出了跨芯片扩展的 2D-mesh，并为每一个跨芯片的 2D-mesh 通道单独提供流控。

图 5-8 是跨芯片逻辑扩展的 2D-mesh 结构图。其中蓝色的互联线是上一小节介绍的跨芯片的环形网络；与环形网络直接相连的是上一小节介绍的 2D-mesh 与环形网络的适配器 BDIOM，BDIOM 不仅用于 2D-mesh 到环形网络协议转化，而且也负责为跨芯片的 2D-mesh 每一个物理通道提供流控（子板和母板上的 BDIOM 之间）。红色虚线标识的是通过环形网络和 BDIOM 逻辑上跨芯片互连的 2D-mesh 数据通路。当这样使用环形网络时，需要将环形网路设定成点到点的通信方式，例如 n5 仅仅可以与 n11 通信、n6 仅仅和 n10 通信。如图 5-8 所示，在子板上实现了 2x2 的 2D-mesh 网络，在母板上也实现了 2x2 的 2D-mesh 网络，这样通过环形网络和 BDIOM 互连在逻辑上系统中实现了一

个 4x2 的 2D-mesh，这将可以集成 8 个通用处理器核。

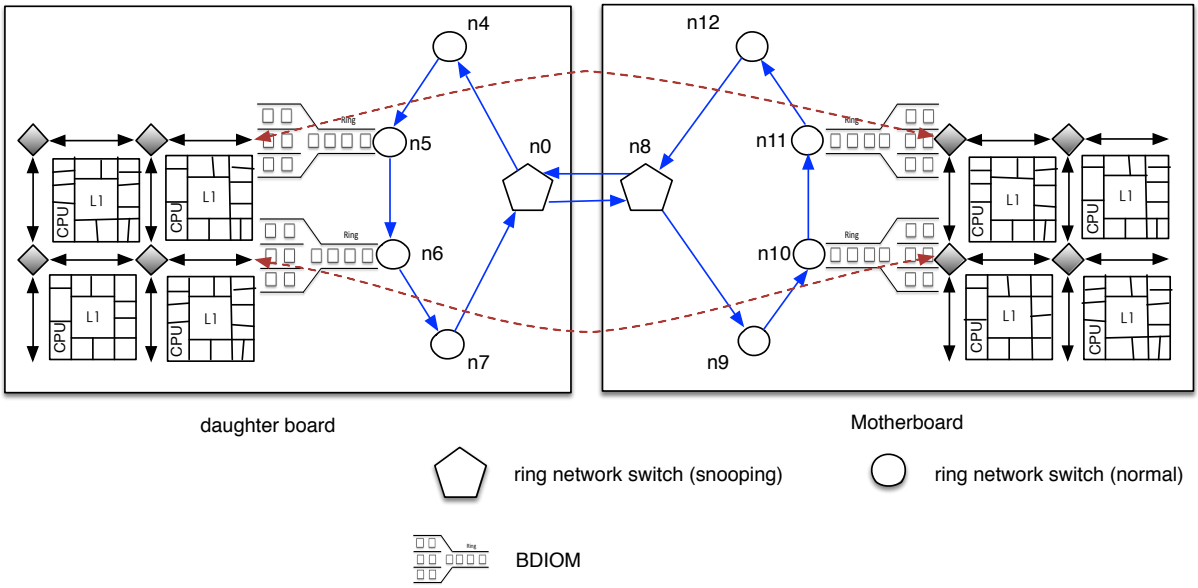


图 5-8 跨芯片逻辑可扩展 2D-mesh

目前设计使用环形网络连接了两块 FPGA 芯片中的逻辑用于验证将要 28nm 工艺流片的 ASIC 芯片。团队其他成员也开发了 PCB 板用于之后使用 FPGA 和 ASIC 来共同验证下一个版本的流片 ASIC。

5.1.5 基于通道的 PCI Plug 设计

Basejump 调试成功并可以提供的外设控制器包括：串口控制器（RS232 接口）、网络接口控制器、DDR3 控制器、DDR2 控制器和 PCIE 控制器。其中串口控制器较为简单，网络接口控制器可以直接使用赛灵思（Xilinx）IP，本文将不再介绍。DDR3 控制器和 DDR2 控制器也使用 IP 核，并添加从 2D mesh 到 DDR 控制器的数据转换和控制器，可以仅仅使用硬件解决；FMC 控制器也是纯硬件解决方案本文也不再介绍。PCI 的控制器则较为复杂，除了使用 IP 核以外，还要设计自己的可编程 I/O 空间（PIO），以及设计主机上的驱动程序。另外设计还在 PIO 中实现了一些特殊的寄存器便于上位机查看整个 Basejump 的工作状态，所以本小节将对 PCI Plug 的设计进行介绍。

在 Basejump 中 PCI Plug 的主要用途是使得上位机和开发板可以通过 PCIE 接口连接，并通过这个接口进行通信。通信的数据有以下三种：1）传输程序、数据以及开发板上需要执行的系统调用信息。2）开发板上面硬件模块的状态信息，包括存储器控制器状态、FMC 接口状态等等。3）一些特殊的开发板命令数据包，例如系统刚启动时，上位机可以发送复位信号数据包复位整个硬件系统。此外还有一些测试开发板的命令数据包。

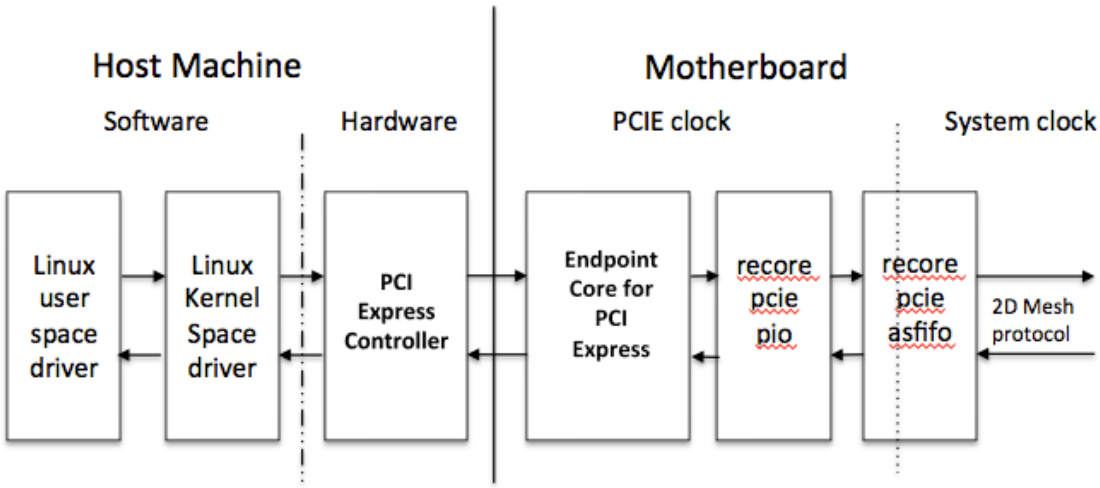


图 5-9 PCI Plug 模块图

图 5-9 是 PCI Plug 的模块图，PCI Plug 分布在上位机和 Basejump 母板上。在 Basejump 母板上，PCI Plug 与 2D mesh 网络直接相连，经过一组用于跨时钟域的异步缓存就连接到可编程 IO 空间，接口底层的控制器直接调用 IP 核。本小节还将介绍 PIO 的设计。在上位机上，硬件部分由服务器主板直接集成，但是需要提供软件驱动程序。

PCI Plug 是直接挂载在 2D mesh 网络的边沿，所以 PIO 的设计也必须支持通道式的通信模式。为了支持这种通道的通信模式，设计在 PIO 的硬件中为所支持的每一个通道添加了 2 个队列（FIFO），接收队列负责缓存从上位机传入到开发板的数据，输出队列负责缓存从开发板输出到上位机的数据。在 PIO 中设计为每一个队列分配一个地址，这样上位机就可以通过读写这些地址来发送和接收开发板的数据。

上位机和开发板之间的通信也是流控的，设计在 PIO 的硬件中为每一个队列添加一个状态寄存器，这些状态寄存器对于上位机来说都是只读的，见图 5-10。其中为接收队列添加的状态寄存器记录了队列中还有多少空闲空间。这样系统启动后，驱动程序读取这些接收队列状态寄存器初始化驱动程序内部的计数器，然后就可以连续向开发板发送数据直到计数器为零，此时重新读取硬件 PIO 的接收状态寄存器，并赋值给软件计数器，如果此时计数器不为零则可以继续发送数据。这样就保证了从上位机到开发板可以快速、安全的发送数据。在另一个方向上，硬件为 PIO 的发送队列添加的状态寄存器记录了发送队列中有多少数据需要发送。这样在系统启动时，驱动程序读取这些发送队列状态寄存器，并初始化驱动程序中软件计数器，然后一次性从开发板读取所有数据到驱动程序中的软件队列，当顶层软件读空驱动程序中的软件队列后，驱动程序读取硬件的发送队列状态寄存器并更新驱动中的计数器，并一次性读取所有数据到软件。为了保证所有从开发板发送的数据都被读取，在程序执行结束前，驱动程序也要轮询硬件的发送状态寄存器，保证不丢包。

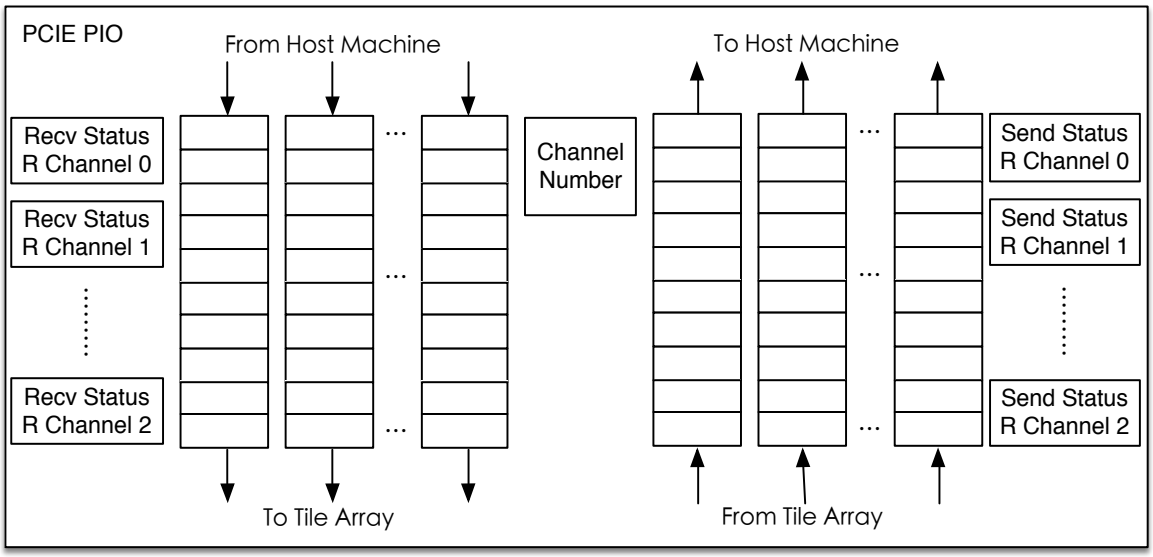


图 5-10 支持多通道模式通信的 PCI Plug 结构

表 5-1 PIO 中地址分配

Channel Regs	add_i[5:4]	add_i[3:0]	Description
receive_status	2'b00	i	记录接收队列中有多少空闲空间
receive_fifo	2'b01	i	接收队列
transmit_fifo	2'b10	i	输出队列
transmit_status	2'b11	i	记录输出队列中有多少有效数据
Special Regs	Address (Linux)	Address (PIO)	
channel_number	0x7fc	0x1ff	存储了通道数量，只读
host_reset	0x7f8	0x1fe	当写入 0xffff_ffff 时，生成 500 周期的系统复位信号，只写
test_reg	0x7f4	0x1fd	用于测试的寄存器，可以写入一个值再读回，可读写
status_reg	0x7f0	0x1fc	硬件状态寄存器，只读
			i：为通道编号

除了前面提到的这些硬件队列和队列状态寄存器，设计在 PIO 中还提供了一些特殊的 32 位寄存器。通道数量寄存器（channel number）记录了 PCI Plug 中一共有多少个通道，系统刚启动时驱动程序读取这个寄存器用于初始化软件中的队列列表。上位机复位寄存器（host reset）提供了一种软件复位整个开发板的方法。系统启动后软件可以向这个寄存器写入约定的数据包 0xffff_ffff，此时硬件就会产生一个 500 时钟周期的复位信号用于复位整个开发板（母板和子板所有电路）。测试寄存器（test reg）用于测试通信

是否正常，驱动软件可以先向这个寄存器写入一个值，然后再读回并检查。状态寄存器（status reg）用于检查开发板上面所有需要检查的硬件部件的状态，例如 DDR 控制器的状态、FMC 接口的状态、系统的启动状态等等。驱动程序通过读取这个寄存器来了解硬件是否准备好工作。

在 PIO 中设计为前面提到的每一个寄存器都分配一个地址，具体的地址见表 5-1。目前本设计的 PIO 最多只支持 16 个通道。Linux 系统上的地址是分配的 IO 空间地址的低位地址。

上位机驱动软件分为系统内核级的驱动和用户空间驱动两部分。系统内核级的驱动主要负责在系统中注册设备、为开发板分配 IO 地址空间和系统内核空间以及与硬件底层相关的通信流控。用户空间驱动将内核地址空间映射到用户空间，并且提供了一些读写硬件队列的封装函数，这些函数的底层就是对映射后的地址指针的读写。更高层级的通道控制程序和运行时环境可以直接调用这些封装好的函数。

5.1.6 系统复位和启动

前文大体上分模块介绍了 Basejump 中较为重要模块的设计，由于系统较为复杂，并且跨越多块 PCB 板和芯片，本小节介绍系统是如何复位和启动的以进一步说明系统各个模块之间的关系。

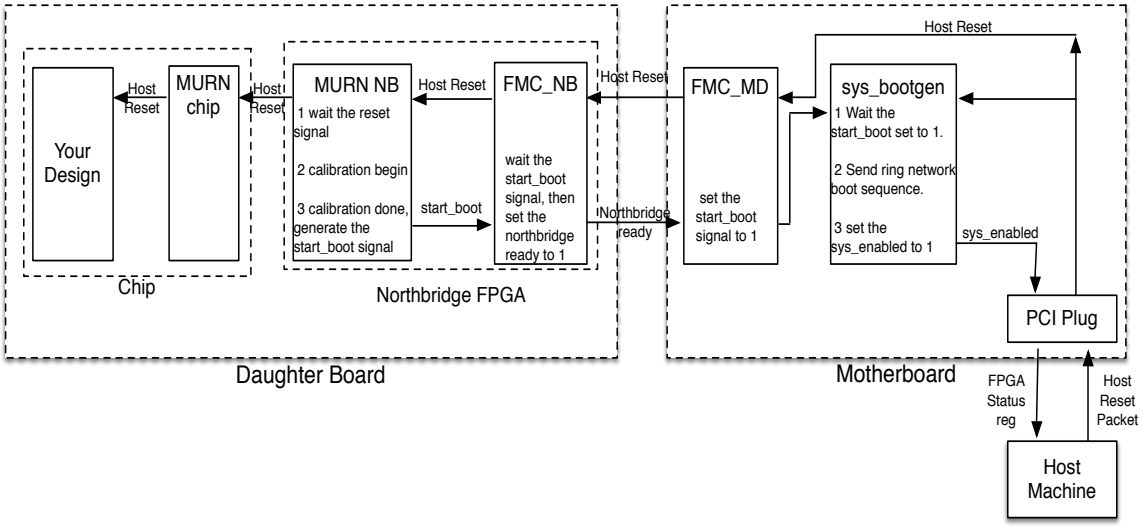


图 5-11 Basejump 系统复位和启动过程

图 5-11 是系统复位和启动的过程演示图。系统的启动过程大体上分为以下步骤：

1）上位机发送复位数据包给 PCI Plug，当 PCI Plug 的 PIO 接收到复位数据包时将生成复位信号用于复位整个硬件系统。此时上位机开始轮询 PIO 中的硬件状态寄存器，直到系统完全启动并且所有状态均正确。

2）sys_bootgen 模块用于生成环形网络命令，当它接收到复位信号后则开始等待 FMC 模块发送回来的开始启动（start_boot）信号。这个开始启动信号表明所有 FMC 左

侧的模块都已经准备好了。

3）当 FMC_MD 收到复位信号时，它要将这个复位信号从母板传输到子板，同时开启 FMC 从母板到子板的校准过程。

4）当北桥的 FMC_NB 接收到复位信号后，复位 FMC_NB 所有电路并将这个复位信号传送给 MURN_NB。

5）MURN_NB 接收到复位信号后，将信号传输给用户定制芯片（chip），之后开启 MURN 的校准过程。当 MURN 的校准过程结束后，发送一个信号给 FMC_NB。

6）FMC_NB 发起 FMC 从子板到母板的校准过程。校准完成后 FMC 发送给 sys_bootgen 开始启动信号。

7）接收到开始启动信号后，sys_bootgen 开始生成环形网络命令，打开需要测试的设计节点。之后 sys_bootgen 置位硬件状态寄存器中的系统已启动信号。

8）此时上位机轮询到的硬件系统状态为准备好，至此系统完全启动好。之后上位机可以开始传送程序和数据给设计并进行计算。

5.2 CoDA 原型系统

设计一种全新的处理器架构不仅仅需要搭建软件的仿真器来分析性能和验证功能正确性，还需要真实地使用 RTL 代码设计电路，并且使用 FPGA 进行验证，这样才能保证架构是硬件可以实现的，进一步的才可以进行芯片的设计。由于 FPGA 芯片上的资源有限，无法一次性验证集成成百上千个专用处理器的 CoDA 架构，所以本小节使用最多两块 FPGA 一起来验证集成了 20 多个专用协处理器的简单 CoDA 架构 GreenDroid。更多的专用协处理器可以通过每次集成不同的专用协处理器来分批进行验证。

上一小节介绍的 Basejump 可以帮助架构师快速的建立原型验证系统，本小节就将 GreenDroid 作为设计集成到 Basejump 中来搭建验证环境。首先展示原型系统实现中的细节，例如 RTL 代码、工具使用情况、原型的搭建步骤等等；之后将展示目前搭建好的原型系统；最后讲述整个验证的过程，包括模块测试、完整的应用程序测试以及回归测试方法等。

5.2.1 原型系统搭建

GreenDroid 的原型验证系统较为复杂，所以有必要简单介绍一下与平台系统搭建过程有关的信息，包括 RTL 代码准备、开发工具的使用情况和其他搭建原型系统时解决的问题。

各种源代码的静态统计见表 5-2 和 5-3。通常情况下，表格中统计的是代码的行数，其中硬件模块的代码统计的就是 Verilog 和 SystemVerilog 的行数。总的来说，GreenDroid 原型系统由 40181 行 RTL 源代码组成，这些代码中不包括 C-core 代码（C-core 的代码不需要手工编写，直接由工具生成）以及调用的 IP 核代码（工具自动生成）。编写 RTL 代码时，本文已经对可以参数化的代码进行了较为彻底的参数化，并且对复杂的通信接

口使用了类似结构体的代码，减少了代码行数也减少了人为代码出错的可能，进而可以大幅度减少调试时间。此外，为了实现这个系统还需要 143 行 TCL 脚本和 343 行 Makefile。其中 TCL 脚本是用于从 RTL 到生成二进制 FPGA 下载文件以及下载所需的所有工具的脚本，Makefile 是为了使所有这些过程可以自动化完成所需要的脚本。这 2172 行 Python 代码大部分用于调试功能，本文用 ChipScope 将感兴趣的信号从 FPGA 导出到文件中，使用这些 Python 脚本将这些信号的 0/1 数据翻译成可以理解的各种数据并进行分析。还有 Python 脚本对时序报告中的多时钟周期路径进行分析，此外还有 Python 脚本用于将所有 C-core 自动集成到 GreenDroid 中。2818 行 C/C++代码是运行在上位机上面的各个层级的驱动程序，这些代码是本文为 GreenDroid 新写的代码，运行时环境代码大部分使用 MIT Raw 处理器的运行时环境代码所以没在统计之内。这些新编写的 IO Master 驱动程序使用了 C++中的继承和多态技术，可以支持 PCI、串口和网口通信，通过这种方式大大的减少了代码量。

表 5-2 FPGA 原型验证系统各个模块 RTL 代码量

模块	行数	字数	字符数	字数占比
Main processor - Control	8998	28538	321422	22.96%
FPU	3104	11349	92806	9.13%
Static Network	3175	9433	108366	7.59%
Main processor - Datapath	2590	7731	94774	6.22%
Data Cache	1947	5954	69429	4.79%
I/O Ports	2039	5701	57262	4.59%
Dynamic Network	1693	5163	64131	4.15%
Test Network	703	1998	17463	1.61%
Integer Divider	457	1101	10700	0.89%
Datapath Module Wrappers	3526	11936	134185	9.61%
GreenDroid Top Level Glue	234	697	8885	0.56%
FPGA Top Level Glue	1359	3652	56476	2.94%
Virtual Tile Array	528	1893	24168	1.52%
FMC adapter	2450	8016	89069	6.45%
BDIOM	1411	4653	61471	3.75%
Ring Network	372	1132	12567	0.91%
MURN	3113	7918	95208	6.37%
PCI Plug	1000	2882	40802	2.32%
Memory Plug	1482	4522	63110	3.64%
Total	40181	124269	1422294	100%

作者尽力的将可以参数化的代码都进行了参数化，这样就可以通过设置不同的参数来构建各种不同的设计，例如包含不同瓦片数量的设计、不同 Cache 配置的设计、不同规模 2D-mesh 互连网络的设计等等。对于瓦片阵列，本文在 FPGA 上测试过 1x1、2x1、2x2 以及 4x2 阵列。阵列使用的 2D mesh 网络边缘也可以通过参数配置是否连接外设，是否需要插入虚拟终端外设（用于保证边缘流控正常）。环形网络的交换节点数量也是

可以通过参数配置的。此外外设中可以配置的东西也做到了参数化，比如 PCI Plug 支持的通道数量，生成复位信号周期；FMC 接口宽度等等。类似这部分参数化的代码大多数都使用 SystemVerilog 编写，因为相对于 Verilog 它可以更好的支持参数化。对于复杂的接口，使用 Verilog 定义了一些数据类型的结构体，如图 5-12。不使用 SystemVerilog 提供的接口（interface）是因为实现时发现目前接口还不能很好的被所有 FPGA 和 ASIC 流程的工具全部支持。

表 5-3 搭建完整原型验证系统所需的各种脚本和软件代码量

文件种类	行数	字数	字符数
TCL 脚本	143	472	4199
Makefile 脚本	343	927	11202
Python 脚本	2172	6178	62884
Total	2496	7172	73362
Linux 内核驱动	712	1703	17622
Linux 用户空间驱动	311	1055	8911
IO Master 驱动	1795	5868	50505
Total	2818	8626	77038

由于在代码编写过程中较多的使用了 SystemVerilog，所以 FPGA 开发流程的综合工具采用了 Synopsys 提供的 Synplify。综合后再将网表导入 Xilinx 提供的开发工具。本文使用 Makefile 自动的分步骤调用每一个需要使用的开发工具，这些调用的工具不但对硬件进行了从 RTL 到 FPGA 二进制下载文件的综合、布局布线等等步骤，而且还对所有软件进行了编译。有了完全自动化的工具链，作者就可以对配置不同参数的多个不同系统同时进行综合（提交到多个不同服务器），例如包含不同数量瓦片的阵列，或者包含不同数量或不同种类的 C-core，这样可以大大加快验证的速度。

```
typedef struct packed {
    logic [3:0]        valid;
    logic [3:0] [31:0] data;
    logic [3:0]        thanks; // going in same direction, but for symmetric channel
} port4_sif;
```

图 5-12 复杂接口的结构体定义

流程自动化、代码的参数化以及最大限度的代码复用，大大减少了设计各种模块的开发时间，减少了配置不同系统的时间以及构建系统和测试系统的时间。这些保证了我们这 4~6 个研究生组成的团队可以开发这样的大型系统以及承担后面介绍的流片工作。作者甚至还编写了脚本监控服务器，当服务器上任务运行完成时给提交任务的用户发送邮件，这样甚至减少了需要设计师坐在电脑前的时间。

5.2.2 GreenDroid FPGA 原型

原型系统的开发也是一步步逐渐进行的，最开始的时候设计使用一块包含 Virtex 5 的 XUPV5 开发板进行原型系统开发，这块开发板主要验证了存储控制器以及包含一个瓦片和几个 C-core 的 GreenDroid。由于 Virtex 5 资源不足，后来的开发就迁移到了包含

Virtex 6 的 ML605 开发板，该开发板集成一块 Virtex-6 XC6VLX240T FPGA 芯片，具有约 240K 逻辑单元和 14976KB RAM 块。开发迁移到 ML605 开发板的时候，开始仅仅使用一块开发板并且将所有的逻辑都放在一块 FPGA 芯片上。这个单开发板的系统可以测试上位机、存储器、PCIE 以及用户自己的设计。之后作者使用了两块开发板并用 FMC 排线将两块开发板连接起来，这样就可以搭建较为完整的原型系统。

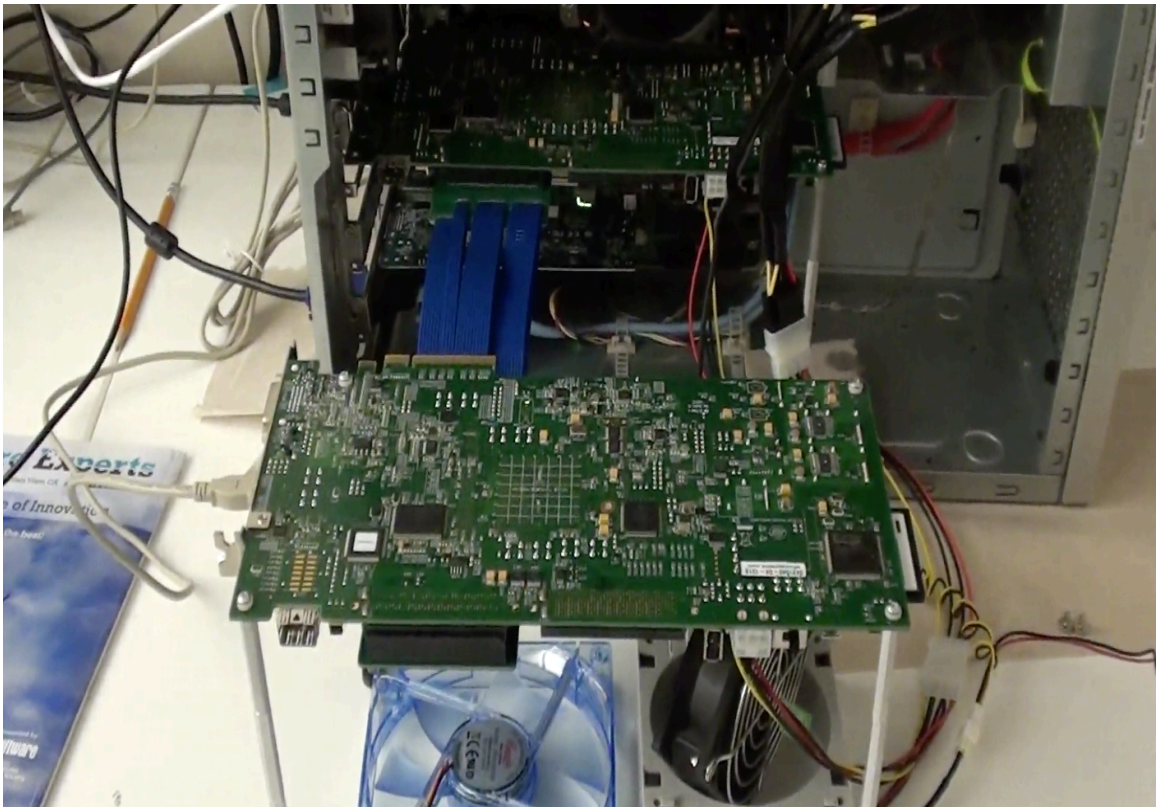


图 5-13 GreenDroid 原型系统图

图 5-13 是 GreenDroid 原型系统的照片。上位机是一台 Sun 的服务器，上面安装的是 CentOS 5.8 操作系统。通过 PCIE 接口插入上位机的 ML605 开发板是 Basejump 中的母板。实物图上的母板上面实现了图 5-2 中模块框图中母板上的所有模块。FMC 排线通过 FMC 的 HPC 连接两块开发板，目前 FMC 中的 LVDS 信号对可以稳定运行在 400MHz。这个目标的实现基于作者仔细地对时钟域和 FPGA IO 资源进行了较好的约束。放在上位机机箱外面的开发板是子板，目前子板采用与母板相同的开发板。子板上实现了图 5-2 中模块框图中子板上的所有模块，除了 MURN IO 间的互连也实现在一块芯片中而不是连接两块芯片。为了在硬件中较为真实的模拟跨芯片的真实情况，本文为不同的 MURN 通道设置了不同的工作时钟频率。子板为了使用 FMC 排线连接母板需要拆除开发板上 FPGA 的风扇，所以为了散热方面的考虑需要在子板下面加装外置的风扇。由于两个开发板需要协同工作，所以为了保证工作电压稳定两块开发板的供电全部由主机上的电源提供。

团队其他成员也设计了两块 PCB 板，一块 PCB 板上包含了两块 Xilinx FPGA 芯片，

其中一块下载北桥芯片，一片下载需要流片的电路。另一块 PCB 板就是包含一块 FPGA 芯片（北桥）和一个全定制的设计芯片。这两块 PCB 板上可以通过 FMC 接口直接插入母板，将不再需要排线。未来的工作就是测试这两块 PCB 板。

5.2.3 系统验证

原型验证系统搭建成功后就需要准备测试程序。目前的测试程序分成两种，一种是为了测试某些特定模块而设计的模块测试程序；另一种是为了测试整个系统平台而从基准测试程序中抽取的标准测试程序。

表 5-4 模块测试程序统计

测试模块	测试程序个数	测试模块	测试程序个数
alu	3	host	12
static network	1	integer	11
cache	10	interrupts	7
control transfer	6	io mux	21
Dcache miss	5	memory counter	19
dynamic network	14	pins	1
event counters	20	instructions	26
floating point	10	streaming dram	1
Total			167

表 5-4 列出了模块测试程序，这些测试程序有些是使用 MIT Raw（类似 MIPS）汇编编写，其他一些是用 C 程序编写。有一些模块测试程序是功能性的测试，例如 alu 的测试；有一些测试是时序性质的，例如 Cache 测试中对缓存缺失周期数的测试。表 5-5 中列出的是所有使用的标准测试程序。这些程序可以较为有效的验证整个系统，包括所有的硬件模块、所有的外设、上位机上的所有软件、C-core 等等。这些测试程序都是从 SPEC 2000、SPEC 2006、JPEG Group 2000 和嵌入式处理器测试程序中抽取的。

表 5-5 标准测试程序列表

程序名称	程序名称	程序名称
autocorrelation	djpeg	radix
bitmnp	fbital	rgbcmy
btree	fft	rgbhpg
bzip2	Hello world	rgbyiq
cjpeg	host	viterbi
conven	mcf	vpr

通过以下几个方面的修改，使得验证过程可以完全自动化。首先，向体系结构中添加了一组测试网络，测试网络的一端集成到处理器的流水线中，另一端可以通过一系列的通信接口在上位机的终端上打印出测试网络的数据。其次，向瓦片中的主处理器指令集中添加了几条向测试网络注入数据的指令。这些指令可以告诉设计师程序的运行是正确的还是错误的，并在程序所有的检查点都运行正确的时候输出程序执行结束。最后，在所有的测试程序中都添加了校验程序运行是否正确的检测代码，并可以将检查的结果

输出。经过这些修改后，设计师可以将运行所有测试程序的任务一次提交服务器，并设置每一个测试程序都将输出打印到文件，再编写脚本处理这些运行报告文件并生成所有测试程序的测试报告。通过这个报告就可以一目了然的了解多少测试程序通过了，多少测试程序运行失败，哪个运行程序运行失败。通过使用这些方法，可以在系统中添加一些定时任务并通过阅读最后的测试报告来自动的对系统进行测试。

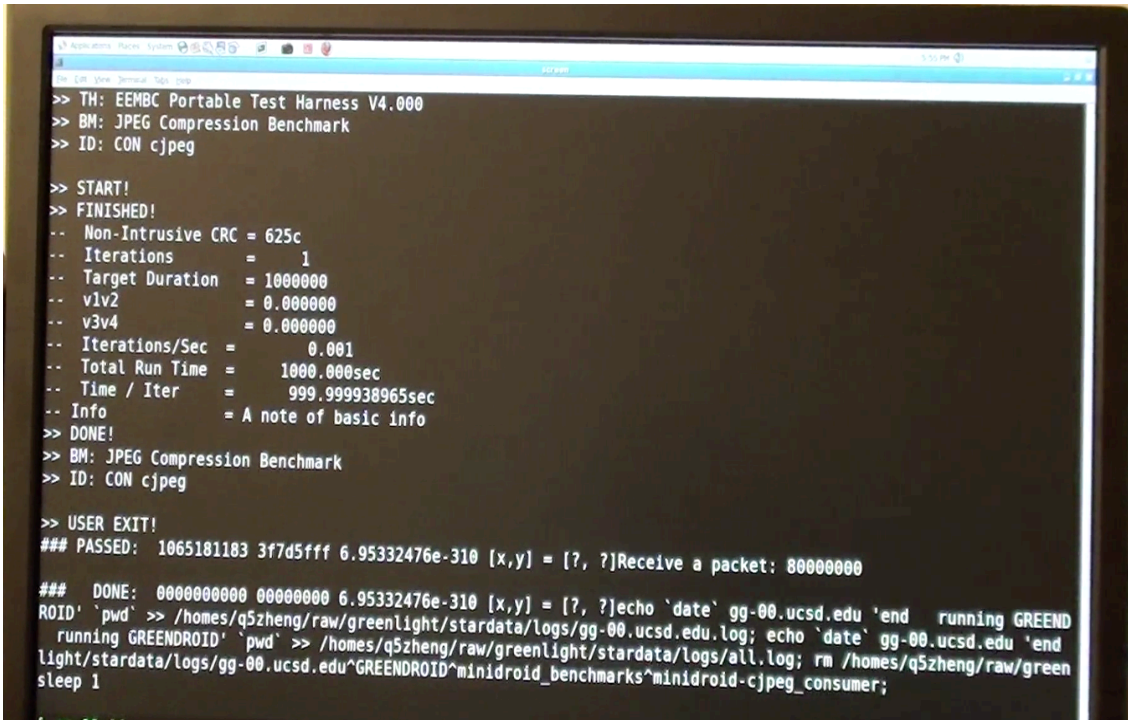


图 5-14 FPGA 验证运行 cjpeg 输出显示

对系统的测试验证包括 RTL 的仿真、FPGA 的验证以及 ASIC 流程网表的仿真。本文所构造的 RTL 仿真（simulation）环境可以仿真所有的 Basejump 模块，包括所有自己编写的 RTL 代码、各种 IP 以及芯片间的互连 MURN IO。作者在 MURN IO 不同的信号对之间插入随机的延迟来模拟 PCB 板上由于布线所造成的不同延迟。通常 RTL 仿真仅仅使用模块测试程序和较为简单的系统测试程序，这是因为全系统 RTL 的仿真速度较慢，一个较大的 SPEC 程序需要运行几天。

对于大型系统测试程序设计师需要使用 FPGA 原型验证平台进行测试。在 FPGA 上对系统进行调试较为复杂，需要猜测出错位置，并向 FPGA 上插入 Chipscope 调试模块进行调试。所以最好的系统调试方法就是尽量地使得 RTL 仿真的代码和 FPGA 上测试的硬件电路代码高度一致，并在仿真中尽量多的解决问题。图 5-14 是在 FPGA 上运行 cjpeg 程序的输出，输出中的 PASSED 和 DONE 就是通过测试网络打印出的消息，PASSED 表示程序运行正确，DONE 表明程序运行结束。由于 FPGA 资源的限制，无法一次性将所有的 C-core 都下载到 FPGA 上进行测试，所以对 C-core 的测试基本上是分次进行的。例如第一次测试向系统中添加了 MCF 的三个 C-core，下一次添加 bzip2 所

需的 C-core。另一个测试是使用跨芯片扩展的 2D-mesh 将逻辑划分到 2 块 FPGA 上，这样就可以一次性地在系统中集成所有需要流片的电路，这是作者验证将要 28nm 工艺流片的 GreenDroid 的主要方法。

我们评估发现单块 Virtex 6 FPGA 芯片仅仅可以实现 4 通用核系统，资源的限制主要来自于 Block RAM。实现具有最小 Cache（32KB I-cache, 32KB D-cache）的 4 核 RAW 处理器，就使用了约 87%的 Block RAM。

在 ASIC 流程中的仿真也是全系统的仿真，只不过芯片部分使用综合后的网表替换，其他部分还是使用 RTL 代码。本文也编写了大量的 Makefile 来支持测试自动化，使得可以仅仅调用不同的 make 目标就可以调用不同类型的测试，例如 make snakeboot 调用 RTL 仿真而 make greendroid 就调用 FPGA 的验证，是进行 RTL 的仿真还是门级网表的仿真也是通过 Makefile 进行配置的。

因为测试过程可以完全自动化进行，所以作者也相应的设计了回归测试的自动化流程。每天晚上指定的服务器会自动的从代码库中 Checkout 出一份最新的代码并进行编译，综合等等。之后开始进行自动化测试，将 RTL 的模块测试任务提交给服务器，并最终生成测试报告；对于 FPGA，在上面顺序运行系统测试程序并生成报告。之后使用脚本将测试报告发送到邮箱。这样使得团队所有成员可以及时发现新提交代码中的问题，并快速进行修改，这样大大提高了开发的效率。

5.3 CoDA 芯片设计

硅实现是检验结构设计的唯一标准。本文所述设计也在进行着两块芯片的流片工作，其中一块芯片是采用 Global Foundry 28nm 工艺流片的 GreenDorid，它包括 4 个瓦片并集成 20 多个 C-core；另一块芯片将通过 MOSIS 使用台积电（TSMC）250nm 工艺流片，这块芯片是一块测试芯片仅仅包含 1 个瓦片；两块芯片中的其他外围电路都是相同的，包括环形网络、MURN I/O 等等。由于访问权限的缘故，本小节使用台积电 250nm 工艺探索实现包含 2 个瓦片的简单的 CoDA（S-CoDA），其中一个瓦片包含通用 MIT Raw 处理器和一个 C-core（支持 autocorrelation），另一个瓦片为第二章介绍的 S-core。MIT Raw 处理器所使用的 2D-mesh 中的静态网络对流数据支持较好，所以 S-core 挂在到静态网络上。这个系统是验证多种不同类型处理器协同工作的最简系统。本小节首先简单介绍一下这个工艺所提供的资源和目标使用的封装；之后分析 S-CoDA 芯片的资源需求；最后将进行芯片设计工作。

5.3.1 芯片资源分析

工艺所能提供的资源，各种资源的属性（延迟、面积和功耗）影响着最终芯片的设计。用硅实现芯片的时候，首先需要查看这些资源和属性是否会对芯片的架构和微结构产生影响。反过来对芯片速度、面积和功耗的需求也会影响对工艺库中资源的选取，例如在 GF 28nm 工艺芯片的流片过程中，为了满足面积的约束最终使用了 7-track 的标准

单元而没有采用默认的 12-track 标准单元。

芯片的资源大体上可以用互连线、逻辑门和管脚来表示。表 5-6 较为详细的列出了 S-CoDA 芯片的主要参数。下面本文分别从互连线、逻辑门和管脚方面进行分析。

为了实现 S-CoDA，本文选用了 TSMC 250nm 工艺最多可以提供的 5 层金属。实现较大规模的 CoDA 架构，需要使用较新的工艺，通常新工艺会提供更多的互连线资源（金属层数），例如台积电 28nm 工艺可以提供 11 层金属互连线。这些金属线对芯片的实现带来两方面的约束：首先，金属线是有方向的。本节选取工艺中 5 层金属线，其中奇数层都是水平的（M1，M3 和 M5）而偶数层是垂直的（M2，M4）。由于金属线几乎不是水平就是垂直的，那么从芯片上一点连接到另外一点在需要拐弯的时候就需要使用不同层次的金属线，这就需要在金属层之间使用通孔（vias）将他们连接起来。这也是第三章计算模块之间距离为什么使用麦哈顿距离的原因。金属层所带来的另一方面的约束是由于顶层金属一般较厚电阻小，所以适合分配全局信号。在这个工艺中，标准单元使用了绝大多数的 M1，所以跨越标准单元的互连线将无法使用 M1。对于存储器（SRAM 或者 Register File）占用从 M1 到 M4 四层金属（取决于存储内部是否需要电源环），所以放置存储器的位置上面仅仅只有 M5 层金属可以自由使用。M5 和 M4 是顶层金属，通常用于电源和地，也就是电源网格（power grid）。此外还要将全局时钟和复位信号布置在较为顶层的金属层中。

表 5-6 S-CoDA 芯片可用资源和封装提供的管脚资源

芯片参数	
Die 尺寸	6 mm x 10 mm
金属层数	5
封装管脚数	356
门资源	
芯片尺寸（Die）	6 mm x 10 mm
逻辑尺寸（芯片尺寸减去放置 I/O 使用的面积）	5.46mm x 9.46mm
可以使用的标准单元行数	1379
每行可以放置的 nand2 数量	1778
芯片可以容纳的最大逻辑等效门数（nand2）	2.5M
线资源	
总金属层数	5
可以自由使用的水平金属层（M5, 部分 M3, 部分 M1）	2+
可以自由使用的垂直金属层（部分 M4, 部分 M2）	1+
管脚资源	
封装所支持的总管脚数	356
Die 可以放置的 Pad 数量	约 680
信号 I/O	96
内核电压	2.5V
IO Pads 电压	3.3V

芯片面积的确定是根据前端综合的面积报告，估算出所有逻辑大概的面积。然后选取使用率（utilization）的经验值 0.7，估算出芯片的面积。这个估算的面积还要根据后

端布局布线后的信息进行修改，并确定最终合适的芯片面积。另一方面，由于设计包含两个瓦片，按照瓦片结构多核处理器设计思路，决定芯片采用长方形的形状。由于顶层金属是水平的，为了生成更好的电源网络，所以设定芯片的形状为如图 5-15 的垂直放置长方形。芯片的面积、形状和电源网格生成后，就可以确定芯片中包含多少个可以放置标准单元的水平行，并由此可以估算整个芯片可以容纳的最大等效门数。

对于封装，S-CoDA 仅仅为了探索芯片实现，所以选取了与本实验室 250nm 流片的另一块芯片同样的封装。选取的是一个包含 356 个管脚的 BGA 封装，并使用焊线（wire bond）将封装和 Die 连接。本文采用后面介绍的锯齿交错的方式来排列 I/O Pad，这样每放置一个 Pad 大概需要 40 微米（um）的空间。通过使用 ICC 的格尺测量芯片的上下两侧每一侧大约有 4800um（封装要求 Pad 距离角端点要有一定的距离）可以放置 I/O，那么上下两侧每一侧大约可以放置 120 个 I/O Pad，上下两侧一共可以放置 240 个 I/O Pad。左右两侧每一侧大概可以有 8800um 可以用于布置 I/O Pad，那么每一侧大约可以放置 220 个 I/O Pad，左右两侧一共可以放置大约 440 个 I/O Pad。总共芯片的四周最多大约可以放置 680 个 I/O Pad。这个数量是远远多于封装所提供的管脚的，这需要在进行封装时将功能相同的两个甚至多个 I/O Pad 用金属线 bonding 到一个 Pin 上。这种方式是提高芯片供电能力的常见方法。

5.3.2 S-CoDA 资源需求分析

表 5-7 S-CoDA 标准单元使用统计

Cell 名	数量	Cell 名	数量	Cell 名	数量	Cell 名	数量
ADDF	5507	CLKBUF	14423	MXI2	1637	OAI221	4542
ADDH	1408	CLKINV	48427	NAND2	21832	OAI222	598
AND2	19343	DFF	6930	NAND2B	1158	OAI2BB1	11172
AND3	175	DFFNS	16	NAND3	2466	OAI2BB2	6632
AND4	3757	DFFR	67	NAND3B	1683	OAI31	172
AOI21	2780	DFFRHQ	29492	NAND4	3041	OAI32	6
AOI211	584	DFFS	19423	NAND4B	2	OAI33	7
AOI22	53408	DFFSHQ	563	NOR2	8388	OR2	1623
AOI221	403	DFFTR	1471	NOR2B	4502	OR3	110
AOI222	10276	EDFF	8313	NOR3	1842	OR4	997
AOI2BB1	2691	EDFFTR	949	NOR3B	109	TLAT	3008
AOI2BB2	85	INV	2215	NOR4	1390	TLATN	62
AOI31	224	JKFF	1	NOR4B	6	TLATR	64
AOI32	117	JKFFS	20	OAI21	9509	XNOR2	317
AOI33	9	MX2	68	OAI211	4385	XOR2	719
BUF	253	MX4	2	OAI22	46041	Total	371420

在进行了布局布线以及生成了时钟树之后，本文对网表中使用的标准单元进行了统计。表 5-7 是 S-CoDA 中使用的标准单元的统计。统计的过程没有对不同驱动能力的标准单元进行区分。统计结果表明设计一共使用了 37 万多个标准单元。表 5-8 对非逻辑运算功能的标准单元进行了数量统计。其中用的最多的是触发器，这些触发器有的用于

构成 MURN I/O 通道内部的 FIFO（太小，不适合使用 SRAM），有的用于各种控制的状态机的状态和各种寄存器。Buffers 中的大部分都用于时钟树的生成，用于平衡时钟树各个分支的相位。芯片中拥有较多的加法器，这些加法器占用了 1.86%的标准单元数量。这些加法器大部分来自于 S-core 中密集的运算阵列。与传统的设计观点不同，本文在设计中为了追求较低的功耗而故意的使用了锁存器（latch），这些锁存器用于环形网络和 MURN 电路中充当流水线中的存储器件。多选器（muxes）中的大部分用于各种片上网络（2D mesh， Ring Network）。

表 5-8 非逻辑功能标准单元的使用情况统计

门类型	数量	百分比
flip-flops （D, JK）	67245	18.10%
buffers	63103	16.99%
adder （ADDF，ADDH）	6915	1.86%
latch	3134	0.84%
inverters	2215	0.60%
muxes	1707	0.46%

表 5-9 列出了主要模块所占用的芯片面积，占用的百分比以及等效的逻辑门数。等效门数的计算采用面积除以 NAND2 的面积的方法计算。整个 S-CoDA 芯片的等效逻辑门数约为 230 万门。其中第三章所介绍的 S-Core 占据了一半以上的面积，其次是第二章所介绍的 GreenDroid 的单核。由于 BDIOM 中有较大的用于流控的队列，所以面积也较大。剩余的其他较大部分为各种互连网络。

表 5-9 S-CoDA 主要模块电路面积以及等效门数

模块名	面积（um^2）	等效门数	百分比
Total	39560368.19	2289374	100%
S-core	24382026.27	1410997	61.6%
GreenDroid (Single Core)	8632978.303	499594	21.8%
bdiom	3173956.318	183678	8%
2D-mesh	1441960.91	83447	3.7%
MURN IO	475453.4471	27515	1.2%
Ring Network	222387.8423	12870	0.6%
Other	1231605.1	71273	3.1%

S-CoDA 所需要的 I/O Pad 包括以下三部分：信号 I/O Pad、芯片内核供电 I/O Pad 以及 I/O 供电电源 I/O Pad。信号 I/O Pad 大致包括以下几个：差分的时钟输入信号（2 个）、复位信号（1）、MURN I/O（22x4，88）和 JTAG(5 个)，共计 96 个信号。对于芯片内核供电 I/O Pad 所需的数量可以进行估算。首先根据功耗评估报告，S-CoDA 的功耗约为 10W，那么在 2.5V 的供电电压下需要的电流至少为 4A。根据工艺库的说明，这个工艺一个管脚可以提供的电流是 25mA，那么就可以计算出需要至少需要 160 个管脚供电，加上地线一共大概需要 320 个 Pad。I/O 供电的电源 Pad 大概为信号管脚的八

分之一，那么也就是说大概需要 13 个管脚为 I/O 提供电源，加上地线一共大概是 26 个左右。特殊的 Pad 有芯片四个角需要放置的拐角 Pad（corner）。将以上的估算相加，大体可以估算出 S-CoDA 需要的管脚数量为 254 个，远远小于 S-CoDA 的芯片可以提供的管脚数量。

5.3.3　芯片实现

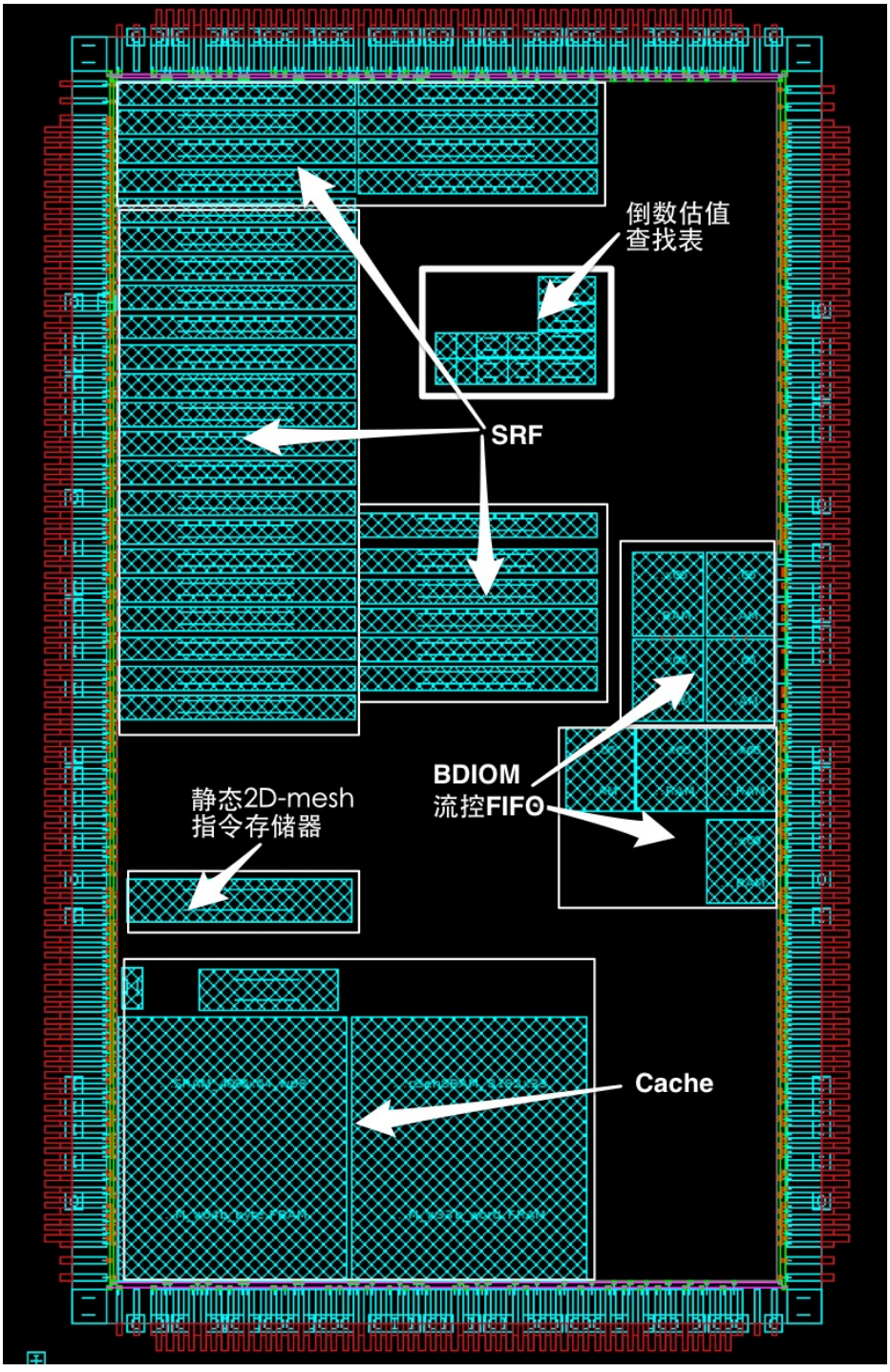


图 5-15 芯片的硬核 IP 布局

芯片的前端使用了 Synopsys 公司的 DC，将 RTL 代码转化为门级网表（.v 文件用

于仿真，.ddc 文件用于后端设计）。本文针对门级网表不但使用 VCS 进行了仿真，同时也使用 Formality 进行了形式化验证。仿真的过程和前一节提到的过程类似，而形式化验证不但要验证 RTL 和 Verilog 格式的门级网表是否一致，还要验证 Verilog 格式的门级网表是否和 DDC 格式的门级网表一致。

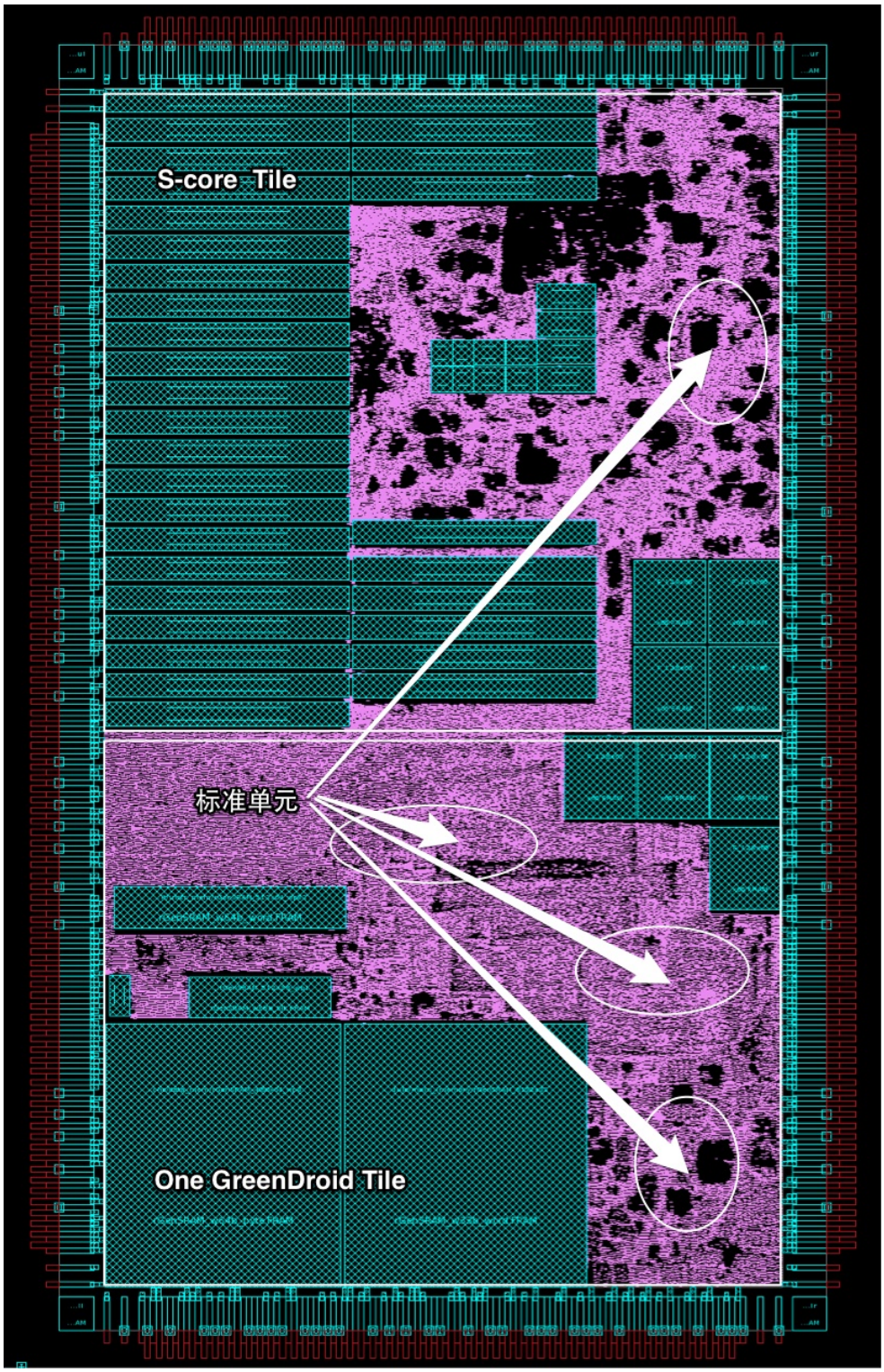


图 5-16 芯片的硬核和标准单元布局

当通过验证后，就可以开始准备后端的工作。本文使用 ARM 提供的存储器 IP，并使用 Artisan 生成所有需要的存储器，这些 IP 包括：SRAM 和寄存器文件（Register File）。之后使用 Milkway 将各种工艺库转化为 ICC（IC Compiler）需要的库。本文使用 ICC 对芯片进行了布局，对所有使用的硬核布局见图 5-15。

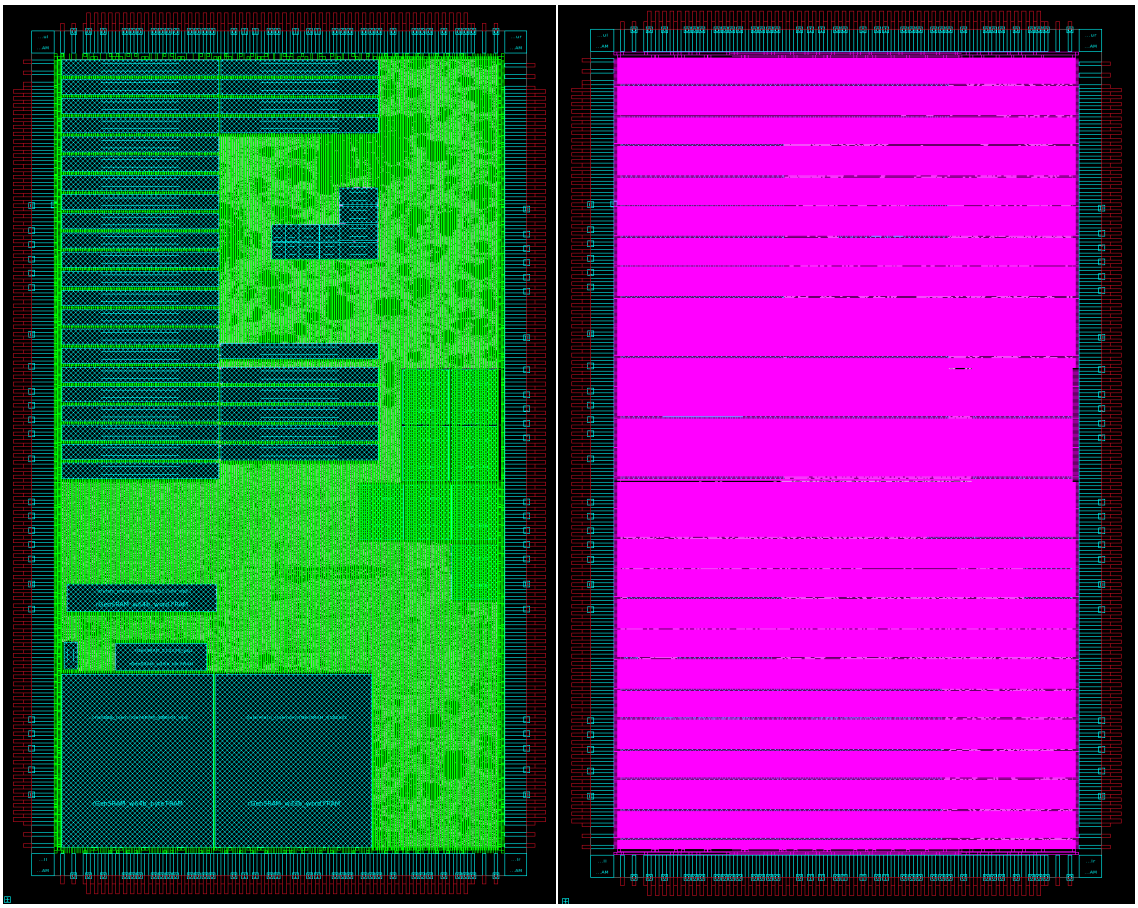


图 5-17 芯片布局后四层（左）和五层（右）金属

大体上整个电路包含 2 个瓦片，上半部分是 S-core，下半部分是一个 GreenDroid 瓦片。为了使 2 个瓦片的面积类似，本文将 S-core 中的流寄存器文件容量减半。上半部分的 32 个 32bit x 128 的 SRAM 组成了 S-core 的流寄存器文件，被这些 SRAM 包围在中间的 12 个小 SRAM 是做浮点除法时候，估计倒数值所需要的查找表（详细算法见第二章）。右侧 8 个 128 x 66 的寄存器文件是 BDIOM 中作为流控的 FIFO。这些 SRAM 周围将布局环形网络和 2D mesh 网络。下半部分的 SRAM 是 GreenDroid 中的主处理器中的存储。图 5-16 在 5-15 的基础上又显示了 M1，这些 M1 表征了标准单元的布局。图 5-17 显示了 M4 和 M5 的金属，可以看到这两层金属中的大部分都被用作电源网格。

对于 I/O 的设计首先需要明确的问题是：设计是 I/O 约束的还是逻辑约束的。I/O 约束的设计一般是逻辑较为简单但是对 I/O 的需求较多，通常这种芯片设计的面积就取决于 I/O 的数量，芯片面积是为了保证可以布置下所有的 I/O。逻辑约束的设计通常是逻辑电路较为复杂但对 I/O 需求较少，这样芯片的面积主要由逻辑决定。虽然 S-CoDA 的

芯片面积属于逻辑约束的，但是为了探索芯片实现，本文还是打算选取稍微复杂的 I/O 设计方法。

对于 I/O pad 的设计，设计规则规定了 bond 之间的距离。对于单圈的 I/O Pad 设计通常有两种方法，见图 5-18。其中一种是平行排列的 I/O(inline bonding)，这种排列方式适合前面说的逻辑约束的芯片。采用这种方式为了遵守设计规则，那么必须在 Pad 之间保留一定的空间，这样才能使得 Bond 之间有足够的空间。另一种方式就是锯齿交错排列的(stagger bonding)，采用这种方式通过选取长短 bond 并交错排列的方式，就可以满足设计需求并不需要 Pad 之间留有较大空隙。S-CoDA 的 I/O 设计就采用了这种方式。对于 I/O 约束的设计如果这种方式还无法满足，还可以采用双圈的 I/O 设计。当采用这种设计的时候，Die 上布下的 I/O Pads 往往可以多于封装所提供的 Pin 的数量，这个时候就可以采用将 2 个或者更多的相同作用的 Pads 连接到同一个 Pin 上。

S-CoDA 采用了锯齿交错排列的 I/O 布局方式，这样可以布下更多的 I/O Pad，这些多余的空间被用于布置更多的电源和地，以争取可以生成更稳定的电源环和电源网格。最终的 Pad 数量分配情况是信号使用了 96 个 Pad，拐角使用了 4 个 Pad，为内核供电的 I/O Pad 使用了 350 个，为 I/O 供电的 Pad 使用了 150 个。图 5-19 就是最后作者所获得的芯片静态电压降分布图。图片的右侧中的数据表明了每一种颜色所对应的电压降低了多少。其中最差的红色也仅仅只有 9.28mV 的压降。在时钟方面，本文假定的目标时钟为 100MHz，除去多时钟周期路径和异步电路路径，其他所有路径都满足约束条件。

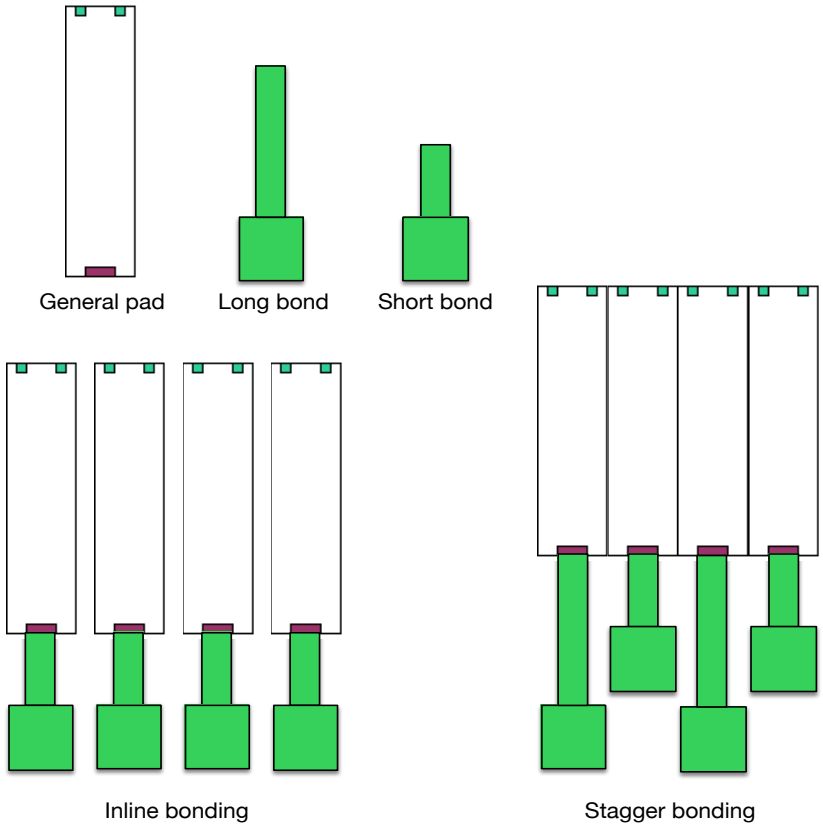


图 5-18 平行排列 I/O 和锯齿交错排列 I/O

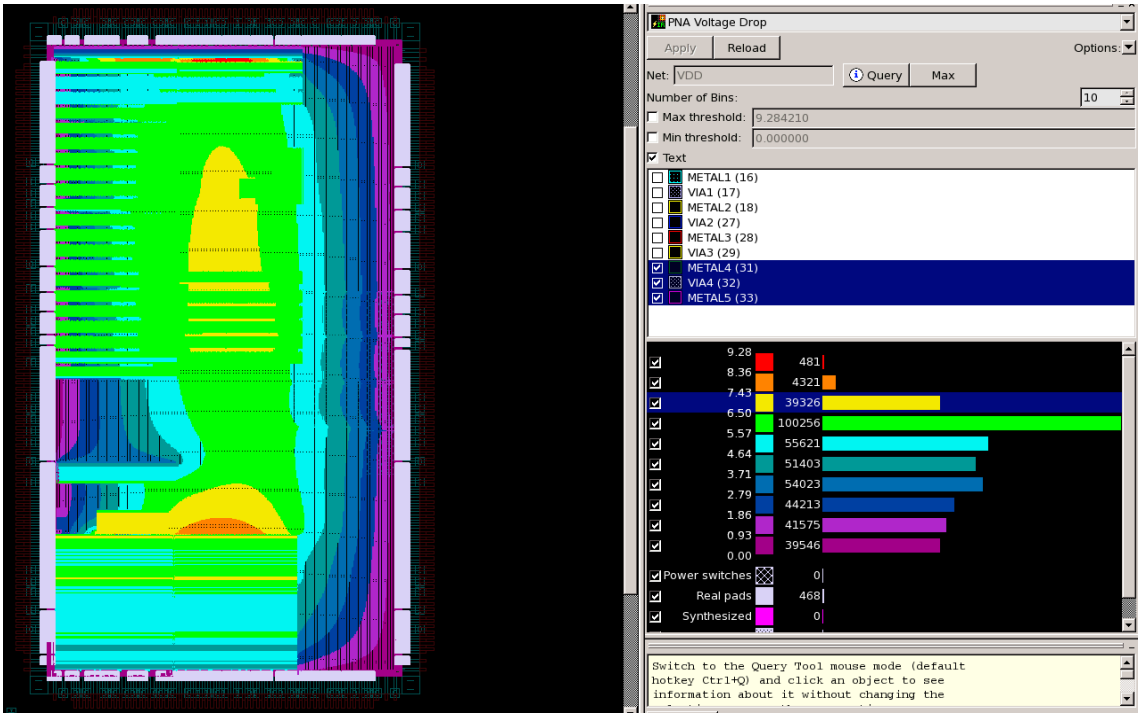


图 5-19 静态电压降（IR Drop）

5.4 小结

由于大量芯片设计团队面临设计原型系统时需要大量时间和精力来集成外围设备的现实问题，本文开发了帮助设计团队快速建立原型的 Basejump 系统。通过将较为简单的 CoDA 设计 GreenDroid 与 Basejump 相连，构建了 CoDA 的 FPGA 原型验证平台。为了一次性验证将要流片的简单 CoDA 架构芯片 GreenDroid 的所有逻辑，本文使用可跨芯片扩展的 2D-mesh 将逻辑划分到 2 块 FPGA 芯片上，并以此搭建了较为完整的原型平台。通过该平台本文验证了整个系统的正确性，包括电路设计的正确性、各种软件的正确性以及最重要的是论证了这种新型的架构是可以正确工作的。之后，尝试性的使用 TSMC 250nm 工艺对简单 CoDA 进行了 ASIC 芯片设计。

本章除了对工程实现方面的介绍，还阐述了一些工程实现中使用的设计方法学。这些方法学包括：模块的大量复用、设计的参数化以及贯穿工程实现始终的自动化。模块的复用和设计参数化的大量使用，首先大大减少了实现整个系统的编码工作量，其次当设计需要修改时也较为方便，最后可以简化电路的调试工作。自动化包括编译综合软硬件代码自动化、测试的自动化、集成的自动化等多个方面。自动化的流程可以最大限度的激发工程师的工作能力，甚至可以使工程师同时进行多项工作；另一方面自动化的流程也可以使得团队新进成员快速融入项目的开发，并使得工具使用的知识快速在团队成员之间传播共享。上面这些设计方法学的成功运用是小团队开发大项目的关键技术基础。

通过 FPGA 原型系统以及团队其他成员 28nm 芯片设计工作，我们可以确定本文所提出的 CoDA 架构是可以工程实现的，并且可以按照设计正确工作。此外，简单的 CoDA 架构芯片可以由几个研究生完成开发，说明这种架构的设计复杂度不是特别高较为容易

推广。

本章所介绍的 Basejump FPGA 设计实现部分由本文作者独立完成，Basejump 中 FMC、PCB 以及芯片封装设计由团队其他成员完成。原型系统设计由本文作者独立完成。本文作者还参与了真实流片项目的前端设计和验证，并独立完成本章介绍的 S-CoDA 的后端设计。

6 结束语

金属氧化物半导体场效应管(Metal-Oxide-Semiconductor Field-Effect Transistor, MOSFET)的亚阈传导漏流是这种器件的本质特征，无法改变。随着工艺尺寸收缩，静态功耗逐渐成为主要能量消耗来源。这样在功耗墙限制下，芯片设计遇到了使用墙问题和暗硅现象，并由此芯片设计进入了暗硅时代。另一方面，目前流行的同构多核架构的扩展方式也将无法继续提升系统的性能和能量效率，这种现象在近期的移动设备上表现明显。各个厂商对设备的比拼已经从应用处理器(AP)的性能、功耗转移到外观设计和主板工艺等等外延技术。这种种现象说明体系结构设计需要突破性的创新。

本文认为未来的体系结构设计对系统能量效率的关注将超过对性能的关注。专用协处理器将是一个很好的选择。在新近工业界所出现的产品中已经开始集成越来越多的传统专用协处理器，这说明向系统中集成专用协处理器是一个可行方案。为了覆盖程序更多的执行部分以及减少专用协处理器开发成本，团队开发了自动生成专用协处理器的C-core 技术。为了使得专用协处理器覆盖更多程序的运行部分，设计师不断向架构中集成越来越多的专用协处理器。最终，系统架构变成了 CoDA 架构，该架构中专用协处理器的数量远远超过通用处理器。本文对 CoDA 架构进行了较为早期的研究工作。

6.1 本文工作总结

作为较早的 CoDA 架构建模、设计空间探索、能效问题研究和较早的 CoDA 原型设计和实现，本文紧紧围绕论证 CoDA 架构设计合理性、可扩展性、能量效率、发现解决未来 CoDA 设计实现所遇到的潜在问题，进行了以下几个方面的研究和工程工作：

（1）研究了 CoDA 对应用的适用性，并以此说明 CoDA 适合暗硅时代。本文分析了安卓移动软件栈，发现大部分应用是基于共享原生库和虚拟机的，硬件化这部分软件就可以使得应用的大部分运行在专用协处理器上。之后重点分析了安卓浏览器，并使用硅构造专用协处理器实现了这个浏览器。实验结果表明在 22nm 工艺下 7mm^2 的硅面积用于构造专用处理器就可以覆盖浏览器 90%的运行。使用可接受的硅面积就可以覆盖应用执行，证明了 CoDA 架构适合暗硅时代。

（2）针对快速探索 CoDA 设计空间的需求，提出了 CoDA 架构分析模型，并对本文提出的多维度可扩展 CoDA 架构进行建模。该架构可以由不同数量的瓦片组成，每一个瓦片可以包含不同数量的函数粒度专用协处理器，并且每一个专用协处理器都可以是异构的。分析模型用来评估每一种特定 CoDA 架构的能量、面积和性能；模型参数既包含了高层次的体系结构参数，也包含低层次的电路实现参数。

（3）探索了 CoDA 架构在不同 Cache 配置、瓦片大小、粗粒度能耗管理策略以及晶体管实现等参数下的能量效率问题。在最优化的参数条件下，与通用架构相比小规模 CoDA 设计可以带来 5.3 倍的能量效率优化和 5 倍的能量延时积（energy-delay product,

EDP）优化；而对于支持上百个应用的大规模 CoDA 设计，可以带来 3.7 倍的能量效率优化和 3.5 倍的 EDP 优化。这说明为大规模应用而设计的大规模 CoDA 扩展是有效的。此外，本文发现 CoDA 设计即使采用了激进的能耗管理策略，漏电功耗所占总功耗的比例仍然随 CoDA 规模增大而增大。

（4）探索了并发执行对 CoDA 能量效率的影响。积极的影响是这些同时运行的程序或线程可以分摊漏电功耗等固定的开销，这样可以提高系统的能量效率。消极的影响是，当驱动 CoDA 生成的目标应用集和实际运行的应用集合不匹配时，会造成大量程序竞争某些专用协处理器，系统的平均能量效率将大大降低。本文提出 CoDA 架构集成覆盖多个函数功能的融合 QsCore 来减少竞争冲突。实验表明使用融合 QsCore 的方式，仅仅增加 41%的面积就可以提供 2 倍数量的专用协处理器，并使得非均匀分布负载的能量效率提高 11.1%~22.1%。

（5）针对使用当前工艺实现的 FPGA 模拟下一代工艺实现的 CoDA 芯片时，单个 FPGA 芯片资源不足的问题，提出了跨多芯片可扩展的 2D-mesh 片上网络。该网络由跨芯片的环形网络连接，并为跨芯片的每一个 2D-mesh 物理通道分别提供跨芯片的流控机制。跨芯片的环形网络提供了 ASIC 芯片到 FPGA 以及 FPGA 之间两种可选连接方案。通过使用该设计方案，本文使用两块 Virtex 6 FPGA 芯片首次实现了 CoDA 架构原型系统。

6.2 进一步工作展望

CoDA 技术作为可以有效优化系统能量效率并适应暗硅时代的架构，有潜力成为未来体系结构研究和工程的重点。本文后续的工作也将继续针对 CoDA 架构设计展开：

（1）继续优化 C-core 的结构设计，特别是一些既能降低能耗又能提高性能的方法将继续添加到 C-core 结构中。这需要继续对 C-core 自动生成工具链进行优化。例如，在某些情况下的器件复用、对非关键路径采用资源占用少速度慢的运算器件、尝试某些异步电路等等(电路级)。

（2）对进一步降低 CoDA 架构能耗的体系结构优化进行研究。可以吸收一些传统处理器中既可以提高性能又可以降低功耗的技术，例如同时多线程。此外，由于 CoDA 中 Cache 的静态功耗较高，需要进一步探讨可配置的 Cache 在 CoDA 中使用的可行性。争取找到一种适应暗硅时代的存储系统。对于互连网络，目前模型中动态功耗占据主导，需要研究降低互连网络动态功耗的技术方案，例如编码技术。大量 C-core 带来的静态功耗也较高，采用 C-core 融合或者引入粗粒度可重构逻辑可能会减少 C-core 的数量，相应的降低所有 C-core 整体静态功耗。

（3）目前 CoDA 架构分析评估模型由将近 1 万行 Python 代码构成并且散落在几个文件中，精细度和易修改性都不足。为了促进 CoDA 架构的研究，应该对该模型进行更好的模块化并采用面向对象编程方法重新编码之后开放源代码，使得更多团队可以快速

理解模型的代码，并进行更为细致的建模和更快速的探索更大的设计空间。

（4）随芯片上资源的增多，向 CoDA 架构中集成更多的由安卓系统生成的 C-core，并探索流片后 CoDA 系统中能量效率的提高情况。另外需要探索新的安卓系统架构。

（5）继续探索流处理器架构，使得这种架构具有更为广泛地适应性并提高这种架构的可编程性。探索是否可以将更多样的算法映射在流处理器上，是否可以通过重构流处理器来移除少量 C-core，为系统带来更好的能量效率。此外，还需要对这种采用粗粒度可重构阵列的流处理器架构中能耗的分布进行仔细的评估。

（6）关注和探索其他类型的专用协处理器和专用处理器。未来的系统中应该存在异构的多种不同种类的处理器，并且每种处理器会适应系统中某些应用。

参考文献

[1] G E Moore. Cramming More Components Onto Integrated Circuits[J]. Proceedings Of The Ieee, 1998,86(1):82-85.

[2] R H Dennard, F H Gaensslen, V L Rideout, et al. Design of ion-implanted MOSFET's with very small physical dimensions[J]. Solid-State Circuits, IEEE Journal of, 1974,9(5):256-268.

[3] N Goulding-Hotta, J Sampson, Z Qiaoshi, et al. GreenDroid: An architecture for the Dark Silicon Age[C]. 2012:100-105.

[4] N Goulding-Hotta, J Sampson, G Venkatesh, et al. The GreenDroid Mobile Application Processor: An Architecture for Silicon's Dark Future[J]. Micro, IEEE, 2011,31(2):86-95.

[5] G Venkatesh, J Sampson, N Goulding, et al. Conservation cores: reducing the energy of mature computations[C]. ACM, 2010:205-218.

[6] N Goulding-Hotta, J Sampson, G Venkatesh, et al. GreenDroid: A mobile application processor for a future of dark silicon[C]. 2010.

[7] M B Taylor. A landscape of the new dark silicon design regime[C]. 2014:1.

[8] M B Taylor. A Landscape of the New Dark Silicon Design Regime[J]. Micro, IEEE, 2013,33(5):8-19.

[9] M B Taylor. A landscape of the new dark silicon design regime[C]. 2013:1.

[10] M B Taylor. Is dark silicon useful? Harnessing the four horsemen of the coming dark silicon apocalypse[C]. 2012:1131-1136.

[11] H Esmaeilzadeh, E Blem, R St. Amant, et al. Dark silicon and the end of multicore scaling[C]. 2011:365-376.

[12] N Hardavellas, M Ferdman, B Falsafi, et al. Toward Dark Silicon in Servers[J]. Micro, IEEE, 2011,31(4):6-15.

[13] J Sampson, G Venkatesh, N Goulding-Hotta, et al. Efficient complex operators for irregular codes[C]. 2011:491-502.

[14] N H E Weste, D Harris. CMOS VLSI Design: a Circuits and Systems Perspective[M].Addison-Wesley, 2005.

[15] Semiconductor Industries Association. International technology roadmap for semiconductors. http://www.itrs.net/Links/2012ITRS/Home2012.htm.

[16] Apple Online Shop, Mac Pro processor selection.

http://store.apple.com/us_smb_78313/buy-mac/mac-pro.

[17] A Frumusanu, J Ho. Huawei Honor 6 Review. http://www.anandtech.com/show/8425/huawei-honor-6-review/4.

[18] M Horowitz. 1.1 Computing's energy problem (and what we can do about it)[C]. 2014:10-14.

[19] G Venkatesh, J Sampson, N Goulding-Hotta, et al. QsCores: trading dark silicon for scalable energy efficiency with quasi-specific cores[C]. ACM, 2011:163-174.

[20] F Jia. The Arsenal Tool Chain for the GreenDroid Mobile Application Processor[D]. San Diego: University of California at San Diego, 2013.

[21] V Govindaraju, H Chen-Han, T Nowatzki, et al. DySER: Unifying Functionality and Parallelism Specialization for Energy-Efficient Computing[J]. Micro, IEEE, 2012,32(5):38-51.

[22] V Govindaraju, H Chen-Han, K Sankaralingam. Dynamically Specialized Datapaths for energy efficient computing[C]. 2011:503-514.

[23] L Wu, M A Kim. Acceleration Targets: A Study of Popular Benchmark Suits[C]. 2012.

[24] W Liang, K Skadron. Implications of the Power Wall: Dim Cores and Reconfigurable Logic[J]. Micro, IEEE, 2013,33(5):40-48.

[25] E G Cota, P Mantovani, M Petracca, et al. Accelerator Memory Reuse in the Dark Silicon Era[J]. Computer Architecture Letters, 2014,13(1):9-12.

[26] J Cong, X Bingjun. Optimization of interconnects between accelerators and shared memories in dark silicon[C]. 2013:630-637.

[27] A Raghavan, L Yixin, A Chandawalla, et al. Designing for Responsiveness with Computational Sprinting[J]. Micro, IEEE, 2013,33(3):8-15.

[28] A Raghavan, L Emurian, S Lei, et al. Utilizing Dark Silicon to Save Energy with Computational Sprinting[J]. Micro, IEEE, 2013,33(5):20-28.

[29] A Raghavan, L Emurian, L Shao, et al. Computational sprinting on a hardware/software testbed[C]. ACM, 2013:155-166.

[30] A Raghavan, L Yixin, A Chandawalla, et al. Computational sprinting[C]. 2012:1-12.

[31] N Pinckney, R G Dreslinski, K Sewell, et al. Limits of Parallelism and Boosting in Dim Silicon[J]. Micro, IEEE, 2013,33(5):30-37.

[32] J M Allred, S Roy, K Chakraborty. Dark Silicon Aware Multicore Systems: Employing Design Automation With Architectural Insight[J]. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2014,22(5):1192-1196.

[33] J Allred, S Roy, K Chakraborty. Designing for dark silicon: a methodological perspective on energy efficient systems[C]. ACM, 2012:255-260.

[34] K Swaminathan, E Kultursay, V Saripalli, et al. Steep-Slope Devices: From Dark to Dim Silicon[J]. Micro, IEEE, 2013,33(5):50-59.

[35] A Sampson, J Nelson, K Strauss, et al. Approximate Storage in Solid-State Memories[J]. ACM Trans. Comput. Syst., 2014,32(3):1-23.

[36] A Sampson, J Nelson, K Strauss, et al. Approximate storage in solid-state memories[C]. ACM, 2013:25-36.

[37] H Esmaeilzadeh, A Sampson, L Ceze, et al. Neural Acceleration for General-Purpose Approximate Programs[C]. 2012:449-460.

[38] H Esmaeilzadeh, A Sampson, L Ceze, et al. Architecture support for disciplined approximate programming[C]. ACM, 2012:301-312.

[39] H Jie, M Orshansky. Approximate computing: An emerging paradigm for energy-efficient design[C]. 2013:1-6.

[40] Z Jia, X Yuan, S Guangyu. NoC-sprinting: Interconnect for fine-grained sprinting in the dark silicon era[C]. 2014:1-6.

[41] H Bokhari, H Javaid, M Shafique, et al. darkNoC: Designing energy-efficient network-on-chip with multi-Vt cells for dark silicon[C]. 2014:1-6.

[42] M Shafique, S Garg, J Henkel, et al. The EDA challenges in the dark silicon era[C]. 2014:1-6.

[43] M Haghbayan, A Rahmani, P Liljeberg, et al. Online testing of many-core systems in the Dark Silicon era[C]. 2014:141-146.

[44] T S Muthukaruppan, M Pricopi, V Venkataramani, et al. Hierarchical power management for asymmetric multi-core in dark silicon era[C]. 2013:1-9.

[45] N Clark, A Hormati, S Mahlke. VEAL: Virtualized Execution Accelerator for Loops[C]. 2008:389-400.

[46] Y Kikuchi, M Takahashi, T Maeda, et al. A 40 nm 222 mW H.264 Full-HD Decoding, 25 Power Domains, 14-Core Application Processor With x512b Stacked DRAM[J]. Solid-State Circuits, IEEE Journal of, 2011,46(1):32-41.

[47] J L Gustafson. Reevaluating Amdahl's law[J]. Commun. ACM, 1988,31(5):532-533.

[48] 胡伟武，陈云霁，肖俊华，等. 计算机体系结构[M] 清华大学出版社.

[49] G Balakrishnan, T Reps, N Kidd, et al. Model checking x86 executables with codesurfer /x86 and WPDS++[C]. Springer-Verlag, 2005:158-163.

[50] P Anderson, M Zarins. The CodeSurfer software understanding platform[C]. 2005:147-148.

[51] T Teitelbaum. CodeSurfer[J]. SIGSOFT Softw. Eng. Notes, 2000,25(1):99.

[52] R E Kidd. The OpenIMPACT Whole Program Optimization Framework[D]. Urbana-Champaign: University of Illinois at Urbana-Champaign, 2007.

[53] M B Taylor, J Kim, J Miller, et al. The Raw microprocessor: a computational fabric for software circuits and general-purpose programs[J]. Micro, IEEE, 2002,22(2):25-35.

[54] M Vuletic, P Ienne, C Claus, et al. Multithreaded virtual-memory-enabled reconfigurable hardware accelerators[C]. 2006:197-204.

[55] J H Ahn, W J Dally, B Khailany, et al. Evaluating the Imagine stream architecture[C]. 2004:14-25.

[56] U J Kapasi, W J Dally, S Rixner, et al. The Imagine Stream Processor[C]. 2002:282-288.

[57] J D Owens, S Rixner, U J Kapasi, et al. Media processing applications on the Imagine stream processor[C]. 2002:295-302.

[58] B Khailany, W J Dally, U J Kapasi, et al. Imagine: media processing with streams[J]. Micro, IEEE, 2001,21(2):35-46.

[59] 张春元，文梅，伍楠，等. 流处理器研究与设计[M] 电子工业出版社，2009.

[60] 伍楠. 高效能流体系结构关键技术研究[D]. 国防科学技术大学，2008.

[61] 文梅. 流体系结构关键技术研究[D]. 国防科学技术大学，2006.

[62] 伍楠. 流处理器MASA内核的研究及实现[D]. 国防科学技术大学，2005.

[63] W Thies, M Karczmarek, S Amarasinghe. StreamIt: A Language for Streaming Applications[M].Springer Berlin Heidelberg, 2002.

[64] I Buck, T Foley, D Horn, et al. Brook for GPUs: stream computing on graphics hardware[C]. ACM, 2004:777-786.

[65] P Mattson. A Programming System for the Imagine Media Processor[D]. Palo Alto: Stanford University, 2001.

[66] 许牧. 可重构众核流处理器体系结构关键技术研究[D]. 中国科学技术大学，2012.

[67] 付江平. 浮点单元超越函数的硬件实现及其优化[D]. 西北工业大学，2007.

[68] J E Stine, N Naresh. Compressed symmetric tables for accurate function approximation of reciprocals[C]. 2006:4-802.

[69] 刘华平. 高性能浮点除法及基本函数功能部件的研究[D]. 中国科学院研究生院

（计算技术研究所）， 2003.

[70] D Joseph. Prefetching Using Markov Predictors[C]. 1997:252-263.

[71] M J Schulte, J E Stine. Symmetric bipartite tables for accurate function approximation[C]. 1997:175-183.

[72] M B Taylor, J Psota, A Saraf, et al. Evaluation of the Raw microprocessor: an exposed-wire-delay architecture for ILP and streams[C]. 2004:2-13.

[73] R Jotwani, S Sundaram, S Kosonocky, et al. An x86-64 core implemented in 32nm SOI CMOS[C]. 2010:106-107.

[74] M B Henry, R Lyerly, L Nazhandali, et al. MEMS-Based Power Gating for Highly Scalable Periodic and Event-Driven Processing[C]. 2011:286-291.

[75] M B Henry, L Nazhandali. From transistors to MEMS: Throughput-aware power gating in CMOS circuits[C]. 2010:130-135.

[76] H F Dadgour, K Banerjee. Design and Analysis of Hybrid NEMS-CMOS Circuits for Ultra Low-Power Applications[C]. 2007:306-311.

[77] Standard Performance Evaluation Corporation. SPEC CPU 2006 benchmark specifications. http://www.spec.org.

[78] Standard Performance Evaluation Corporation. SPEC CPU 2000 benchmark specifications. http://www.spec.org.

[79] Eembc benchmark suite. http://www.eembc.org.

[80] C Lattner, V Adve. LLVM: a compilation framework for lifelong program analysis & transformation[C]. 2004:75-86.

[81] S Thoziyoor, N Muralimanohar, J H Ahn, et al. CACTI 5.1. Tech report. http://www.hpl.hp.com/techreports/2008/HPL-2008-20.html.

[82] M Bohr, K Mistry. Intel's Revolutionary 22nm Transistor Technology. http://download.intel.com/newsroom/kits/22nm/pdfs/22nm-Details_Presentation.pdf.

[83] B C Lee, E Ipek, O Mutlu, et al. Architecting phase change memory as a scalable dram alternative[C]. ACM, 2009:2-13.

[84] IMOD Technology Overview. IMOD technology overview. http://www.qualcomm.com/common/documents/whitepapers/QMT_Technology_Overview_12-07.pdf.

[85] M L Alles, H L Hughes, D R Ball, et al. Total-Ionizing-Dose Response of Narrow, Long Channel 45nm PDSOI Transistors[J]. Nuclear Science, IEEE Transactions on, 2014,61(6):2945-2950.

[86] R V Joshi, R Kanj, S Saroop. Novel 4 GHz Interleaved SRAM Cells With Asymmetrical Precharge in 45 nm PDSOI Technology[J]. Semiconductor Manufacturing, IEEE Transactions on, 2014,27(2):278-286.

[87] P Chao, H Zhiyuan, Z Zhengxuan, et al. A New Method for Extracting the Radiation Induced Trapped Charge Density Along the STI Sidewall in the PDSOI NMOSFETs[J]. Nuclear Science, IEEE Transactions on, 2013,60(6):4697-4704.

[88] S Sirohi, S Khandelwal. Modeling low frequency noise in PDSOI MOSFETs for analog and RF applications[C]. 2010:1940-1942.

[89] D Guo, A Bryant, W Xinlin, et al. Gate-dielectric permitivity and metal-gate work-function tradeoff in L_{met}=25nm PDSOI device characteristics[J]. Electron Device Letters, IEEE, 2006,27(6):505-507.

[90] F Abouzeid, A Bienfait, K C Akyel, et al. Scalable 0.35 V to 1.2 V SRAM Bitcell Design From 65 nm CMOS to 28 nm FDSOI[J]. Solid-State Circuits, IEEE Journal of, 2014,49(7):1499-1505.

[91] X Chuan, K Banerjee. Physical Modeling of the Capacitance and Capacitive Coupling Noise of Through-Oxide Vias in FDSOI-Based Ultra-High Density 3-D ICs[J]. Electron Devices, IEEE Transactions on, 2013,60(1):123-131.

[92] J P Noel, O Thomas, M Jaud, et al. Multi-UTBB FDSOI Device Architectures for Low-Power CMOS Circuit[J]. Electron Devices, IEEE Transactions on, 2011,58(8):2473-2482.

[93] J Mazurier, O Weber, F Andrieu, et al. On the Variability in Planar FDSOI Technology: From MOSFETs to SRAM Cells[J]. Electron Devices, IEEE Transactions on, 2011,58(8):2326-2336.

[94] S H Rasouli, K Endo, J F Chen, et al. Grain-Orientation Induced Quantum Confinement Variation in FinFETs and Multi-Gate Ultra-Thin Body CMOS Devices and Implications for Digital Design[J]. Electron Devices, IEEE Transactions on, 2011,58(8):2282-2292.

[95] Y Choi, H Jin-Woo, L Hyunjin. Reliability Issues in Multi-Gate FinFETs[C]. 2006:1101-1104.

[96] G Knoblinger, C Pacha, F Kuttner, et al. Multi-Gate MOSFET Design[C]. 2006:65-68.

[97] M C Johnson, D Somasekhar, C Lih-Yih, et al. Leakage control with efficient use of transistor stacks in single threshold CMOS[J]. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2002,10(1):1-5.

[98] M C Johnson, D Somasekhar, K Roy. Leakage control with efficient use of transistor stacks in single threshold CMOS[C]. 1999:442-445.

[99] K Nii, H Makino, Y Tujihashi, et al. A low power SRAM using auto-backgate-controlled MT-CMOS[C]. 1998:293-298.

[100] F Hamzaoglu, Y Yibin, A Keshavarzi, et al. Analysis of dual-V/sub T/ SRAM cells with full-swing single-ended bit line sensing for on-chip cache[J]. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2002,10(2):91-95.

[101] N Azizi, A Moshovos, F N Najm. Low-leakage asymmetric-cell SRAM[C]. 2002:48-51.

[102] F Hamzaoglu, Y Yibin, A Keshavarzi, et al. Dual-V_T SRAM cells with full-swing single-ended bit line sensing for high-performance on-chip cache in 0.13 μm technology generation[C]. 2000:15-19.

[103] T Bjerregaard, J Sparsoe. Scheduling discipline for latency and bandwidth guarantees in asynchronous network-on-chip[C]. 2005:34-43.

[104] T Bjerregaard, S Mahadevan, R G Olsen, et al. An OCP Compliant Network Adapter for GALS-based SoC Design Using the MANGO Network-on-Chip[C]. 2005:171-174.

[105] T Bjerregaard, J Sparso. A router architecture for connection-oriented service guarantees in the MANGO clockless network-on-chip[C]. 2005:1226-1231.

[106] F Feliciian, S B Furber. An asynchronous on-chip network router with quality-of-service (QoS) support[C]. 2004:274-277.

[107] T Felicijan, J Bainbridge, S Furber. An asynchronous low latency arbiter for Quality of Service (QoS) applications[C]. 2003:123-126.

[108] J Bainbridge, S Furber. Chain: a delay-insensitive chip area interconnect[J]. Micro, IEEE, 2002,22(5):16-23.

[109] J Lee, S Kim. Filter Data Cache: An Energy-Efficient Small L0 Data Cache Architecture Driven by Miss Cost Reduction: Computers, IEEE Transactions on[Z]. 2014: PP, 1.

[110] A Bardizbanyan, M Sj, Lander, et al. Towards a performance- and energy-efficient data filter cache[C]. ACM, 2013:21-28.

[111] Y Chia-Lin, L Chien-hao. HotSpot cache: joint temporal and spatial locality exploitation for I-cache energy reduction[C]. 2004:114-119.

[112] W Tang, R Gupta, A Nicolau. Design of a predictive filter cache for energy savings in high performance processor architectures[C]. 2001:68-73.

[113] M Huang, J Renau, Y Seung-Moon, et al. L1 data cache decomposition for energy

efficiency[C]. 2001:10-15.

[114] J Kin, M Gupta, W H Mangione-Smith. The filter cache: an energy efficient memory structure[C]. 1997:184-193.

[115] Z Chuanjun, F Vahid, Y Jun, et al. A way-halting cache for low-energy high-performance systems[C]. 2004:126-131.

[116] Z Chuanjun, F Vahid, Y Jun, et al. A Way-Halting Cache for Low-Energy High-Performance Systems[J]. Computer Architecture Letters, 2003,2(1):5.

[117] M D Powell, A Agarwal, T N Vijaykumar, et al. Reducing set-associative cache energy via way-prediction and selective direct-mapping[C]. 2001:54-65.

[118] S Kaxiras, H Zhigang, M Martonosi. Cache decay: exploiting generational behavior to reduce cache leakage power[C]. 2001:240-251.

[119] X Cuiping, Z Ge, H Shouqing. Fast Way-Prediction Instruction Cache for Energy Efficiency and High Performance[C]. 2009:235-238.

[120] T Adegbija, A Gordon-Ross, A Munir. Dynamic phase-based tuning for embedded systems using phase distance mapping[C]. 2012:284-290.

[121] Z Chuanjun. A Low Power Highly Associative Cache for Embedded Systems[C]. 2006:31-36.

[122] Z Chuanjun, F Vahid, W Najjar. A highly configurable cache architecture for embedded systems[C]. 2003:136-146.

[123] A Malik, B Moyer, D Cermak. A low power unified cache architecture providing power and performance flexibility[C]. 2000:241-243.

[124] D H Albonesi. Selective cache ways: on-demand cache resource allocation[C]. 1999:248-259.

[125] P Gi-Ho, L Kil-Whan, L Jang-Soo, et al. A low-power cache system for embedded processors[C]. 2000:316-319.

[126] H Jingcao, R Marculescu. Energy- and performance-aware mapping for regular NoC architectures[J]. Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on, 2005,24(4):551-562.

[127] H Jingcao, R Marculescu. Application-specific buffer space allocation for networks-on-chip router design[C]. 2004:354-361.

[128] S Banerjee, N Dutt. FIFO power optimization for on-chip networks[C]. ACM, 2004:187-191.

[129] C Xuning, P Li-Shiuan. Leakage power modeling and optimization in interconnection

networks[C]. 2003:90-95.

[130] L Kangmin, L Se-Joong, Y Hoi-Jun. SILENT: serialized low energy transmission coding for on-chip interconnection networks[C]. 2004:448-451.

[131] C Yu-Liang, L Shaoshan, C Eui-Young, et al. An Energy and Performance Efficient DVFS Scheme for Irregular Parallel Divide-and-Conquer Algorithms on the Intel SCC[J]. Computer Architecture Letters, 2014,13(1):13-16.

[132] A B Kahng, K Seokhyeong, R Kumar, et al. Enhancing the Efficiency of Energy-Constrained DVFS Designs[J]. Very Large Scale Integration (VLSI) Systems, IEEE Transactions on, 2013,21(10):1769-1782.

[133] S Eyerman, L Eeckhout. A Counter Architecture for Online DVFS Profitability Estimation[J]. Computers, IEEE Transactions on, 2010,59(11):1576-1583.

[134] T Yoneda, K Masuda, H Fujiwara. Power-Constrained Test Scheduling for Multi-Clock Domain SoCs[C]. 2006:1-6.

[135] J Schmid, J Knablein. Advanced synchronous scan test methodology for multi clock domain ASICs[C]. 1999:106-113.

[136] C Sitik, B Taskin. Multi-voltage domain clock mesh design[C]. 2012:201-206.

[137] L Souef, C Eychenne, E Alie. Architecture for Testing Multi-Voltage Domain SOC[C]. 2008:1-10.

[138] R Cooksey, S Jourdan, D Grunwald. A stateless, content-directed data prefetching mechanism[C]. ACM, 2002:279-290.

[139] D Joseph, D Grunwald. Prefetching using Markov predictors[J]. Computers, IEEE Transactions on, 1999,48(2):121-133.

[140] N P Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers[C]. 1990:364-373.

[141] S Xian-He, S Byna, C Yong. Improving Data Access Performance with Server Push Architecture[C]. 2007:1-6.

[142] W Hassanein, Jos, Fortes, et al. Data forwarding through in-memory precomputation threads[C]. ACM, 2004:207-216.

[143] L Chi-Keung, H Sunpyo, K Hyesoon. Qilin: Exploiting parallelism on heterogeneous multiprocessors with adaptive mapping[C]. 2009:45-55.

[144] P H Wang, J D Collins, G N Chinya, et al. EXOCHI: architecture and programming environment for a heterogeneous multi-core multithreaded system[C]. ACM, 2007:156-166.

[145] J Owens, D Luebke, N Govindaraju, et al. A survey of general-purpose computation on

graphics hardware[C]. 2005:21-51.

[146] J Nickolls, I Buck, M Garland, et al. Scalable parallel programming with CUDA[C]. ACM, 2008:1-14.

[147] M Mishra, T J Callahan, T Chelcea, et al. Tartan: evaluating spatial computation for whole program execution[C]. ACM, 2006:163-174.

[148] K Sankaralingam, R Nagarajan, L Haiming, et al. Exploiting ILP, TLP, and DLP with the polymorphous TRIPS architecture[C]. 2003:422-433.

[149] K Sankaralingam, R Nagarajan, L Haiming, et al. Exploiting ILP, TLP, and DLP with the polymorphous trips architecture[J]. Micro, IEEE, 2003,23(6):46-51.

[150] S Swanson, K Michelson, A Schwerin, et al. WaveScalar[C]. 2003:291-302.

[151] F Hannig, S Roloff, G Snelting, et al. Resource-aware programming and simulation of MPSoC architectures through extension of X10[C]. ACM, 2011:48-55.

[152] J Allred, S Roy, K Chakraborty. Designing for dark silicon: a methodological perspective on energy efficient systems[C]. ACM, 2012:255-260.

[153] S Mingoo, S Hanson, L Yu-Shiang, et al. The Phoenix Processor: A 30pW platform for sensor applications[C]. 2008:188-189.

致　谢

在论文完成之际，在此感谢所有曾经指导、关心和帮助过我的老师、朋友及亲人！

深深感谢我的导师高德远教授。读博期间，在学习、科研中高老师对我要求严格，鼓励我大胆创新，追求卓越；在生活上，高老师也给予了无微不至的关怀和实质性的帮助。高老师严谨的治学态度、对国防事业的鞠躬尽瘁、对学生的宽容和关爱是我终生学习的榜样。高老师的鼓励和关怀，我永远铭记在心。

深深感谢博士生指导小组的樊晓桠教授。樊老师对我悉心教导，从流处理器的研究、学术研究的方法以及论文的撰写都给予了指导。樊老师以渊博的学识和对学术方向高屋建瓴的准确把握，在本文流处理器部分和论文结构组织方面给予了极大的帮助。

深深感谢博士生指导小组的张盛兵教授。张老师以认真严谨、精益求精的工作态度，在迷雾中为我指明了方向。从我本科进入实验室开始，张老师就在学习、科研、论文撰写和生活等诸多方面给予了我无私的指导和真诚的帮助。

深深感谢外导师 UCSD 的 Michael Taylor 教授。Prof.Taylor 对于学术和工程精益求精，不断追求卓越的精神和工作态度感染着实验室每一个人。Prof.Taylor 学识渊博，在包括编译器、操作系统、体系结构、工艺在内的系统架构师所需掌握的各方面对我们进行了全方位指导，甚至传授了我们很多编写代码和组织项目的经验。在您实验室工作的三年时光，永生难忘。

深深感谢外导师 UCSD 的 Steven Swanson 教授。每当遇到学术和工程难题的时候，Prof.Swanson 乐观的态度深深感染了我们。Prof. Swanson 学识渊博，一语中的式的指导方式为我解决了好多疑惑和困扰。

特别感谢宾州州立大学 Jack Sampson 教授。做为 UCSD Arsenal/GreenDroid/Dark Silicon 等项目的早期参与者，Jack 在系统架构高层次设计方面给予我们很多指导。做为项目早期代码的编写者，Jack 像一本百科全书回答了我们所有关于 C-core、QEMU、CoDA 等等方面的代码编写问题。

特别感谢我的课题组长张萌老师，他在科研、生活中给予我很大的指导和帮助，帮我解决了许多实际问题。感谢王党辉老师、安建峰老师、黄小平老师对我的指导、帮助和鼓励。感谢杨颖俊老师、陈超老师、白佳宁老师、小卢老师、国际合作处马颖老师在学习、科研和生活中所给予的帮助。感谢高武、田杭沛、史莉雯和姚涛博士，在科研、学习中，与他们的讨论令我一步步找到了自己的方向；感谢韩立敏、任向隆博士、张晓静博士，在博士学习期间，给予我很多鼓励和帮助。

感谢 GreenDroid 项目合作者 Dr.Ganesh Venkatesh、Dr.Nathan Goulding-Hotta、Joe Auricchio 和 Vikram Bhatt 在设计自动化、系统 Profile、后端设计以及 GreenDroid 架构方面给予的耐心讲解和帮助以及那无数次受益匪浅的讨论。感谢 BSG 项目合作者

Dr.Lu Zhang、Luis Vega 和 Dr. Guan-Lin Wu 在后端设计方面的耐心讲解和帮助。感谢 NVSL 项目组的 Dr.Joel Coburn、Dr.Adrian Caulfield、Dr.Laura Grupp、Trevor、Sudharsan 等在 FPGA 设计、Linux 驱动程序编写和修改 Linux 内核等方面的指点。感谢维护项目组集群的 Dr. Michael Wei、Dr.Sundaram 等，你们的无私奉献保证了我们其他人工作的正常进行。特别感谢 Fei Jia,我们做实验室较早的 2 个中国学生，探讨了很多编译器方面的内容，并且在生活中互相给予了很多帮助和鼓励。感谢所有为 Basejump 付出过心血的同事，感谢体系结构项目组所有同学。

感谢华为海思周昔平博士、夏晶对麒麟处理器架构的讲解。感谢国防科大张春元教授,连续2年的暑期学校也让我受益颇多。感谢胡伟、汤小明、仰凯、贾明明教授、Matrix、沈昊、应亮、Sheng-Yang Hung、武玲娟、唐敏等等所有在美国期间的室友和认识的朋友，我们互相关心和帮助共同度过了充满新奇和陌生的岁月。感谢嵌入式系统集成工程研究中心各个课题组中的同学们，大家朝夕相处，关系融洽，团结协作，形成了很好的学术气氛。在此向刘兴、苏贺鹏、鄢国峰、杨明、马瑞、陈杰、周洪、赵丽丽、任嵘、何向栋等已毕业的硕士表示衷心的感谢。

特别感谢我的父母，从小到大他们给予了我最无私的关爱，为我创造最好的生活和学习条件。

感谢其他曾经给予我关心和帮助的人！

攻读博士学位期间发表的学术论文和参加科研情况

第一作者发表论文：

[1] **Qiaoshi Zheng**，Nathan Goulding-Hotta, Scott Ricketts, Steven Swanson, Michael Taylor, Jack Sampson. Exploring Energy Scalability in Coprocessor-Dominated Architectures for Dark Silicon[J]. ACM Transactions on Embedded Computing Systems(TECS)[1], July 2014, 13(4), 130:1-130:24
(SCI Index, WOS N.O: 000341390100013)
(EI Index, IDS N.O: 20143418087903)

[2] **Qiaoshi Zheng**, Deyuan Gao, Xiaoya Fan, Meng Zhang, Tao Yao and Limin Han. An Evaluation of the Many-core Longtium SP Computer System[C]. The 2013 IEEE International Conference on Signal Processing, Communications and Computing (IEEE ICSPCC 2013), Aug. 2013, Kunming, China, IEEE Computer Society: 1694:1-1694:4 **(EI Index, IDS N.O: 20140417228975)**

[3] **Qiaoshi Zheng**, Deyuan Gao, Liwen Shi, Jie Chen. Tolerating memory latency: L2 cache actively push architecture[C]. The 2009 International Conference on Advanced Computer Theory and Engineering (ICACTE 2009), Cairo, Egypt, ASME: 509-516. **(EI Index, IDS N.O: 20104813425974)**

[4] **Qiaoshi Zheng**, Deyuan Gao, Jie Chen, Chao Wang. AAP and AAPM: improved prefetching structures of the L2 cache[C]. The 2009 International Conference on Advanced Computer Theory and Engineering (ICACTE 2009), Cairo, Egypt, ASME: 569-576. **(EI Index, IDS N.O: 20104813425982)**

非第一作者发表论文：

[5] Nathan Goulding-Hotta, Jack Sampson, **Qiaoshi Zheng**, Vikram Bhatt, Joe Auricchio, Steven Swanson and Michael Taylor. Greendroid: An architecture for the dark silicon age[C]. 17th Asia and South Pacific Design Automation Conference (ASP-DAC), February 2012, Sydney, Australian, IEEE Computer Society: 100-105. (invited paper)**(EI Index, IDS N.O: 20121714967941)**

[6] Vikram Bhatt, Nathan Goulding-Hotta, **Qiaoshi Zheng**, Jack Sampson, Steven Swanson and Michael B. Taylor. SiChrome: A Silicon Browser[C]. 1st Dark Silicon Workshop (DaSi), in conjunction with ISCA, 2012, Portland, USA.

[1] CCF 推荐计算机系统与高性能计算方向 B 类第四的期刊 http://www.ccf.org.cn/sites/ccf/biaodan.jsp?contentId=2567814757412，本文是这期中 Domain-Specific Multicore Computing 专题发表的 6 篇论文中的首篇论文。

[7] 史莉雯，樊晓桠，陈杰，黄小平，**郑乔石**. 程序行为分析指导 TLB 低功耗设计[J]. 计算机科学, 2011, 38(5): 301-305.

[8] Liwei Shi, Xiaoya Fan, **Qiaoshi Zheng**, Jie Chen. Exploiting the Character of Memory Accesses to Achieve Lower Power Consumption of the Data TLB[C]. The 2010 International Conference on Computer Engineering and Technology, April 2010, Cheng Du, China, IEEE Computer Society: V2-271-275. **(EI Index, IDS N.O: 20104313316750)**

[9] Chao Wang, Zhanhuai Li, Xiaoyu Li, **Qiaoshi Zheng**. An error handling method for primary-backup replication protocol[C]. The 2009 International Conference on Advanced Computer Theory and Engineering (ICACTE 2009), Cairo, Egypt, ASME: 569-576. **(EI Index, IDS N.O: 20104813425963)**

[10] Jie Chen, Deyuan Gao, **Qiaoshi Zheng**. A Research on an Optimized Adaptive Dynamic Power Management[C]. The 2009 IEEE International Conference on Computer Science and Information Technology (ICCSIT 2009), August, 2009, Beijing, China, IEEE Computer Society: 52-55. **(EI Index, IDS N.O: 20094612454531)**

[11] 马瑞，张盛兵，**郑乔石**. 一种语音端点检测电路的设计[J].计算机工程与应用，2010, 46(14): 69-74.

获得专利：

[1] 高德远，**郑乔石**，田杭沛，樊晓桠，张盛兵，王党辉，魏廷存，黄小平，张萌，郑然. 嵌入式处理器片内指令和数据推送装置. 发明专利，公开号 101697146A （有效）.

[2] 王党辉，樊晓桠，张盛兵，安建峰，韩茹，张萌，黄小平，陈超，**郑乔石**. 总线监控与调试控制装置. 实用新型，公开号 202267954U （有效）.

获奖情况：

“Altera 杯”第七届全国研究生电子设计大赛

总决赛团体银奖，个人三等奖(个人排名第四) 2010.08，中国北京

西北赛区团体特等奖(第一名)，个人排名第一 2010.06，陕西西安

国防科大“天河计划”暑期学校优秀学员（总成绩排名第一） 2010.07，湖南长沙

国防科大“微处理器设计”暑期学校优秀学员(总成绩排名第二) 2009.08，湖南长沙

参加科研情况：

[1] “系统集成技术研究”（美国国防部先进研究计划局(DARPA)资助下未来架构研究中心(Center for Future Architectures Research, C-FAR)，编号：2384.006）

[2] “Prototyping Platform to Enable Power-Centric Multicore Research” (美国自然科学基金会，编号：1059333)

[3] “Energy-Efficient Parallel Architectures for Computer Vision” （美国自然科学基金会，编号：0846152）

[4] “Arsenal: Extending Moore’s Law through the Design, Synthesis and Use of Massively Heterogeneous Systems”（美国自然科学基金会，编号：0811794）

[5] “面向流计算的主动适应体系结构”（国家“八六三”高技术研究发展计划基金项目，编号：2009AA01Z110）

[6] “高性能片上存储系统”（国家自然科学基金重点项目，编号：60736012）

[7] “片上多核主动适应存储体系结构”（国家自然科学基金项目，编号：60773223）

[8] “Computer Organization and Design——The Hardware/Software Interface，Fourth Edition， David A. Patterson, John L. Hennessy”（国外经典教材）的翻译工作

西北工业大学

学位论文知识产权声明书

本人完全了解学校有关保护知识产权的规定，即：研究生在校攻读学位期间论文工作的知识产权单位属于西北工业大学。学校有权保留并向国家有关部门或机构送交论文的复印件和电子版。本人允许论文被查阅和借阅。学校可以将本学位论文的全部或部分内容编入有关数据库进行检索，可以采用影印、缩印或扫描等复制手段保存和汇编本学位论文。同时本人保证，毕业后结合学位论文研究课题再撰写的文章一律注明作者单位为西北工业大学。

保密论文待解密后适用本声明。

学位论文作者签名：_____　　指导教师签名：_____

年　月　日　　年　月　日

西北工业大学

学位论文原创性声明

秉承学校严谨的学风和优良的科学道德，本人郑重声明：所呈交的学位论文，是本人在导师的指导下进行研究工作所取得的成果。尽我所知，除文中已经注明引用的内容和致谢的地方外，本论文不包含任何其他个人或集体已经公开发表或撰写过的研究成果，不包含本人或其他已申请学位或其他用途使用过的成果。对本文的研究做出重要贡献的个人和集体，均已在文中以明确方式表明。

本人学位论文与资料若有不实，愿意承担一切相关的法律责任。

学位论文作者签名：_____

年　月　日